Information and Computer Science

UNIVERSITY OF CALIFORNIA

Irvine

Learning Problem Solving

A dissertation submitted in partial satisfaction of the requirements for the degree Doctor of Philosophy

UNIVERSITY OF CALIFORNIA
IRVINE

UNIVERSITY OF CALIFORNIA

Irvine

Learning Problem Solving

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy
in Information and Computer Science

by

Bruce W. Porter

Committee in charge:

Professor Dennis Kibler

Professor Richard Granger

Professor Donald Hoffman

1984

The dissertation of Bruce W. Porter is approved,
and is acceptable in quality and form for
publication on microfilm:

Donald D. Hoffman

Richard Gyger

Dennis F. Kibler

Committee Chair

University of California, Irvine

1984

Dedication

This dissertation is dedicated to the memory of my mother,
Dorris Shunk Porter,
whose love and generosity I will never forget.

Contents

List of Tables

List of Figures

Acknowledgments

I am indebted to Dennis Kibler for guidance and support during my years of study. I have enjoyed and benefited from both his active participation in this research and his sincere concern for his students. Also, Rick Granger and Don Hoffman provided useful insights and support.

The involvement of my family was invaluable to me during this undertaking. My parents, James and Dorris, showed me the enjoyment of learning and achieving. Most important of all is the love and support that I received from my wife, Claudia. She helped me through periods of glum and despair. And she was always ready to celebrate the most minor of successes. Thank you all.

Curriculum Vitae

Bruce W. Porter

June 26, 1956	Born Denver, Colorado
1977	B.S. in Computer Science, University of California at Irvine *Magna Cum Laude, Phi Beta Kappa*
1977–1980	Manager of Software Development, Interactive Graphics Division, CALCOMP, Anaheim, California
1980–1981	Teaching Assistant, Computer Science Department, University of California at Irvine.
1981–1982	Research Assistant, Software Reuse Project, Computer Science Department, University of California at Irvine
1982	M.S. in Computer Science, University of California at Irvine
1983–1984	Research Assistant, Machine Learning Project, Computer Science Department, University of California at Irvine
1984	Ph.D. in Computer Science, University of California at Irvine Dissertation: "Learning Problem Solving"

Publications

Kibler, D. and Porter, B. Perturbation: A Means for Guiding Generalization. Appearing in *Proceedings of the International Joint Conference on Artificial Intelligence*, 1983, pp. 415–418.

Kibler, D. and Porter, B. Episodic Learning. Appearing in *Proceedings of the National Conference on Artificial Intelligence*, 1983, pp. 191–196.

Porter, B. and Kibler, D. Learning Operator Transformations. To appear in *Proceedings of the National Conference on Artificial Intelligence*, 1984.

Technical Reports

Kibler, D. and Porter, B. Episodic Learning. Information and Computer Science Department, University of California at Irvine. Technical Report 194, May 1983.

A Survey of Induction Algorithms for Machine Learning. Information and Computer Science Department, University of California at Irvine. Technical Report 207, June 1983.

Fields of Study

Major Field: Computer Science

Studies in Software Engineering

Professors Peter Freeman and James Neighbors

Studies in Artificial Intelligence

Professor Dennis Kibler

Abstract of the Dissertation

Learning Problem Solving

by

Bruce Walter Porter

Doctor of Philosophy in Computer Science

University of California, Irvine, 1984

Professor Dennis Kibler, Chair

Learning to problem solve requires acquiring multiple forms of knowledge. Problem solving is viewed as a search of a state-space formulation of a problem. With this formalism, operators are applied to states to transit from the initial state to the goal state. The learning task is to acquire knowledge of the state-space to guide search. In particular, three forms of knowledge are required: *why* each operator is useful, *when* to apply each operator, and *what* each operator does. A PROLOG implementation, named PET, demonstrates the learning approach in the domains of simultaneous linear equations and symbolic integration.

Episodic learning is a technique for learning why individual operators are useful in a solution path. Episodic learning acquires generalized operator sequences which achieve the goal state. This is done by backing-up state evaluation and learning sub-goals in the state-space.

Perturbation is a technique for learning when individual operators are useful. Perturbation guides the generalization process to discover minimally-constained preconditions for useful operator applications. This is done by experimentation, thereby reducing the teacher's role in the learning process.

Learning **relational models** is a technique for discovering what individual operators do. Relational models are an explicit representation of the transformation performed by operators. This representation enables the learning element to reason with operator semantics to guide further learning.

Episodic learning, perturbation and relational models form an integrated approach for learning problem solving. The approach demonstrates self-teaching by reasoned experimentation.

CHAPTER 1
Scope of the Dissertation

The Main Contribution

The goal of this dissertation is to present a unified approach to learning problem solving. Learning to problem solve is a knowledge acquisition task in which the learner exploits properties of problem solving to facilitate learning. Problem solving can be modelled as a state-space search from an initial state to a goal state. At each step of the search, an operator is selected and applied to the current state to transit to a successor state. The goal of the search is to discover a sequence of transitions which achieve a goal. The learning task is to derive knowledge of the search space by solving one problem which aids in solving other problems. The application of knowledge then replaces search during problem solving.

There are several features of problem solving tasks which facilitate learning. First, goal states are explicit and the learning is goal-directed. This enables the learner to reason about goals to constrain the concepts being learned. Second, problem solving domains are *reactive* [CARB83]. That is, the learner can derive useful information by conducting experiments in the domain. Third, operator sequences frequently recur. This suggests that the learner should try to discover useful sequences and build macro operators. Furthermore, heuristics should be learned which guide the application of these macros.

The contribution of this dissertation is a general approach to efficient learning in problem solving domains and a mechanism which demonstrates the approach. The major features of this contribution are:

1) Episodic Learning – an incremental method for discovering useful operator sequences. Incremental learning requires that knowledge be acquired gradually and be linked with existing knowledge. In problem solving, the learner incrementally acquires knowledge of the search space by constructing useful operator sequences. The purpose of each operator in a sequence is recorded as the achieved sub-goal. Episodic learning is an approach to learning *why* individual operators are useful in problem solving by discovering the role each plays in operator sequences.

2) Perturbation – a technique for automatically generating experiments in the problem solving domain. Experiments are devised which "flush-out" the important lesson of teacher supplied advise. The problem with teacher training is that the advice is too specific. The learner must discover the general lesson. Perturbation partially automates the role of the teacher in learning by discovering *when* individual operators are effective in the problem solving environment.

3) Relational Models – a novel representation for operators and heuristics. Relational models are a representation which makes explicit the transformation performed by an operator. This explicit representation allows the learner to reason with a model of operator semantics to improve the learning rate of domain knowledge. It is unrealistic to assume that "natural" operators have explicit representations. Therefore, an algorithm is presented for learning relational model representations. Relational models are an explicit representation of *what* individual operators do during problem solving.

These contributions are demonstrated with a computer implementation of the learning paradigm in two domains: simultaneous linear equations and symbolic integration.

Why Study Machine Learning?

This section describes the contribution of machine learning to science and engineering. First a definition of learning is proposed which both gives the research a footing in artificial intelligence (AI) and reveals implicit biases. Then, from the issues raised in the definition, the relevance of machine learning to "open" problems in AI and cognitive science is discussed.

A Definition of Machine Learning

First, it is instructive to define the subject of machine learning. Simon [SIMO83] defines learning to be:

> "...any change to a system that allows it to perform better the second time on repetition of the same task or another task from the same population."

This definition correctly implies learning is a dynamic process by which the learner improves performance from experience. However, it confuses the learning mechanism (making changes to a system) with the evaluation of the success of the learning (the performance of the system). These are distinct processes.

Another shortcoming of Simon's definition, as pointed out by Scott [SCOT83], is that it covers activities that we would not want to label as learning. For example, changing the blade in a razor improves performance, but the razor has not "learned." Scott provides his own definition:

> "Learning is the organization of experience."

This definition identifies two important aspects of learning behavior. First, it implies that a learner's experiences potentially effect subsequent behavior. Second,

it implies a dynamic memory [SCHA82] with the ability to store experiences for efficient recall.

A problem with Scott's definition is that it is neither operational nor testable. An operational definition suggests implementations, or properties of implementations. A testable definition is one capable of distinguishing between examples of the definition and non-examples.

Defining learning is as difficult as defining intelligence or expertise. At best, a definition of learning should suggest a framework for exploring the process of learning. My definition of learning which is used in this research is:

> **"Learning is the process engaged in by a learner of building a representation of an environment with the goal of improving problem solving in that environment."**

Implications of the Definition

The definition of learning used in this dissertation gives machine learning a footing in established research areas of AI. First, the definition associates learning with problem solving. That is, the process of problem solving enables learning and learning enables more advanced problem solving. Carbonell [CARB83] makes this association even stronger: "Problem solving and learning are inalienable aspects of a unified cognitive mechanism."

Second, the definition associates learning and knowledge representation. Constructing a representation of an environment consists of two difficult tasks:

1) Selecting a knowledge representation technique.

2) Representing facts and procedures of the environment in terms of the representation technique.

Both of these issues are AI bottlenecks. The first issue was addressed by Amarel [AMAR68] who demonstrated the effect of the problem representation technique on problem solving efficiency. Amarel presented six formulations of the missionaries and cannibals problem. These formulations ranged from problem independent, general-purpose representations to formulations which were highly problem dependent. Formulations which exploit problem characteristics were found to be significantly more efficient for problem solving.

While Amarel demonstrated the power of representation shifts, the problem of *discovering* useful shifts remains. Amarel concluded that this task should be automated. A system which dynamically shifts representation with the goal of improving problem solving is, as defined above, a learning system. Anzai [ANZA78] experimented with shifting strategies during problem solving. Initially, weak problem solving methods, such as breadth-first search and avoiding bad moves, are employed. Gradually, stronger methods such as subgoaling and means-ends analysis are developed from repeated solution of the same problem. Finally, problem-specific strategies like scripts and macros are acquired.

The second issue, frequently called the knowledge acquisition problem, is also important to AI research. Currently this issue is most relevant to the construction of expert systems. Expert systems are computer programs which demonstrate levels of performance comparable to a human expert in a limited problem domain. The task of transferring the expert knowledge from the human to the program is the knowledge acquisition problem in expert systems. The TEIRESIAS system [DAVI77] is an example of the research on knowledge acquisition. The goal of TEIRESIAS was to develop an intelligent assistant to aid the expert in encoding domain knowledge. A set of tools was developed for constructing and maintaining a knowledge base. For instance, a (limited) consistency checker verified that new knowledge did not contradict old knowledge.

However, the knowledge acquisition process is still tedious and errorprone. Human expertise is acquired over decades of both subliminal and explicit training. Paraphrasing Yates, experts embody knowledge but cannot make the knowledge explicit. This suggests an alternate solution to knowledge acquisition. Rather than assisting the expert to express his knowledge explicitly (the TEIRESIAS approach), expert knowledge can be acquired by learning from the expert's problem solving behavior.

Inductive learning programs, which induce general rules from instances of the rules, have proven successful in limited domains. For example, in the domain of diagnosing plant diseases [MICH80], a learning program induced general classification rules from examples of diseased plants classified by experts. These induced rules outperformed rules acquired by "traditional" knowledge acquisition techniques. So an important contribution of machine learning is the construction of mechanisms for the acquisition of expert knowledge.

But perhaps more important than developing mechanisms for learning, research in machine learning identifies principles of intelligence, both human and artificial. Langley and Simon [LANG81] describe learning research as a search for invariants and generality. A goal of science is to understand laws, or invariants, which explain observed phenomena. Systems which adapt to their environment are always changing and no "first-order" laws can account for the system's performance. A theory of learning, not a theory of performance, reveals the invariants.

The goal of the search for generality is to discover parsimonious principles, which are the foundation of a scientific paradigm. For instance, cognitive science has adopted the information processing model as a general principle of intelligence. As recounted by Langley and Simon, the early years of AI focused on discovering general principles which were tested with mechanisms. For example, the General Problem Solver [NEWE61] demonstrated the principle of means-ends analysis for problem solving.

The search for general principles of the 1950's and early 60's gave way to knowledge-intensive, domain-dependent mechanisms of the next generation. Partially driven by an engineering approach to AI, researchers constructed high-performance expert systems which embodied increasing amounts of domain knowledge. The pervasive lesson in this work was that improved performance results from providing more knowledge to a relatively simple process which applies the knowledge. After discovering the importance of knowledge to problem solving, progress was stalled by the knowledge acquisition problem discussed above. Research on learning re-affirms the need for general principles by developing a theory of knowledge acquistion.

This section discussed the relevance of the study of learning to artificial intelligence. The contributions of learning research are two-fold: the construction of mechanisms which can be applied to AI problems and the development of theories which reveal invariants and general principles of intelligent behavior.

Learning Problem Solving

This section discusses the general problem of learning to do problem solving. Questions addressed in this section are:

What class of problem solving tasks can be learned?

What is the advantage of learning?

What sorts of knowledge need to be acquired?

By what technique does the learning occur?

These issues are critical for learning problem solving in any domain. After discussing the general problem in this section, chapters 3–6 address these issues in two specific problem solving domains.

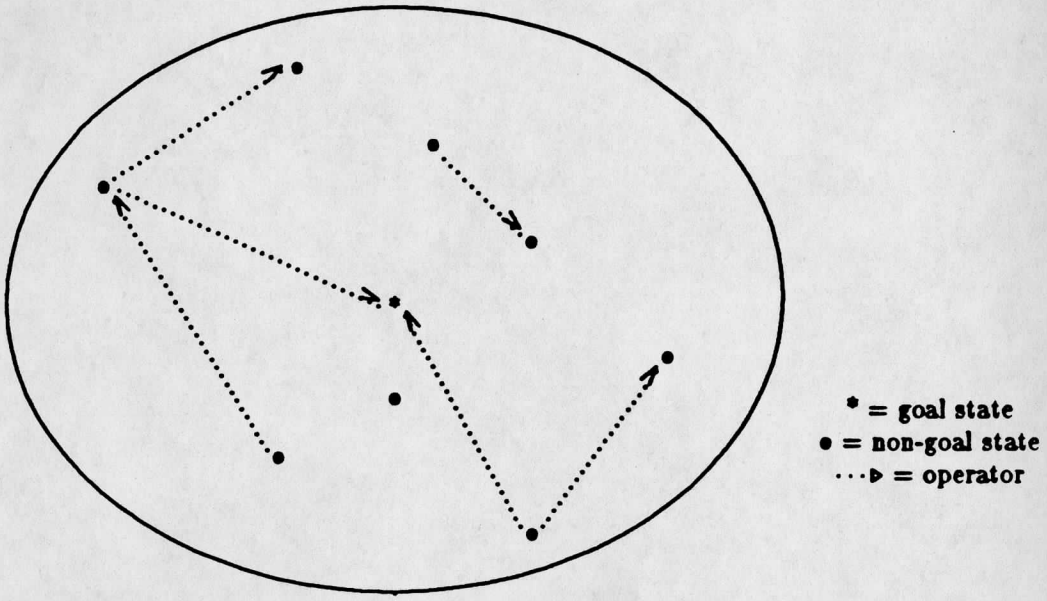


Figure 1

Abstract State Space

Problem Solving as Search

In a weak form, problem solving can be modelled as state space search. Viewed this way, a problem solving task is represented by an initial state (the problem to be solved), a set of goal states (solved problems), and a set of operators which can be applied to solve the problem. This defines a state-space, as shown in figure 1. Problem solving in this model consists of searching for a sequence of operators which transit from the initial state to one of the goal states.

Much research in computer science has focused on efficient methods of searching a state space [NILS80, pp 53–128]. Uninformed search simply explores the search space by random selection of operators to apply. The problem, of course, is that the search space may be so large as to prohibit uninformed search. An alternative is informed search which selects operators to apply based on information gleened from the current state in the search. The operator selection knowledge is embedded in heuristics which may directly recommend an operator, block an operator judged unuseful, or select an operator based on the estimated quality of resulting

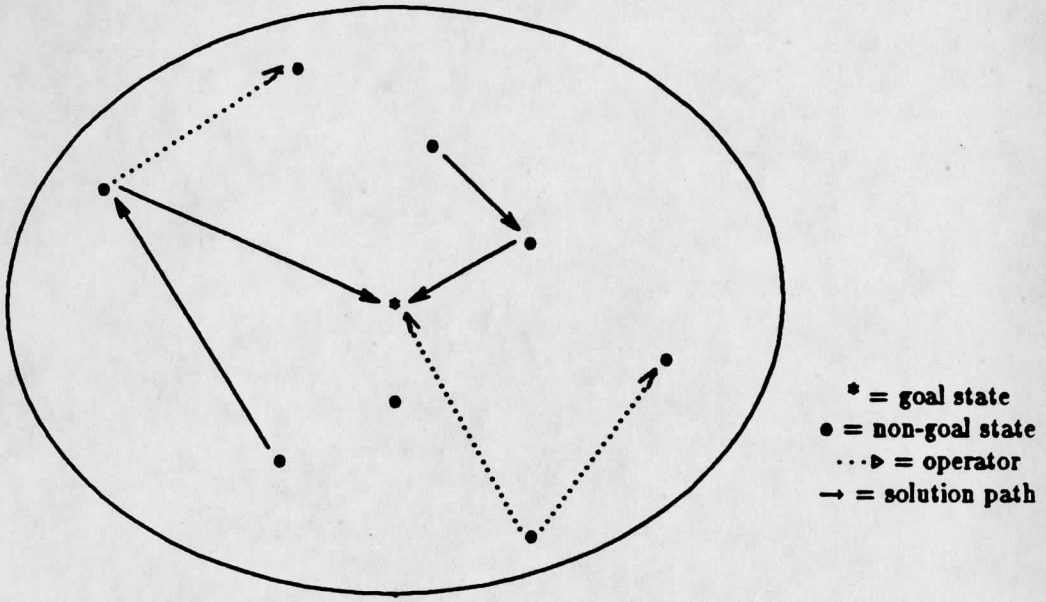


Figure 2

Solution Paths

states. Any task which can be formulated with a state space representation is a candidate for informed search.

The approach to learning problem solving proposed here is to replace search by knowledge. The effect of this on problem solving is that operators are selected to apply to states based on knowledge of the search space rather than by trying alternatives with little domain knowledge to distinguish among them. Specifically, there are three types of search space knowledge needed:

1) Knowledge of Solution Paths – As shown in figure 2, solution paths are sequences of operators which are applied serially. These sequences transit from an initial state to a goal state. Knowledge of solution paths improves problem solving because search is eliminated. The solution path serves as a procedure which dictates the solution with no guess-work. Learning solution paths is the subject of chapter 3 which proposes a technique called **episodic learning**. Episodic learning builds solution paths by learning *why* each operator is useful – i.e., the role of each operator in solution paths.

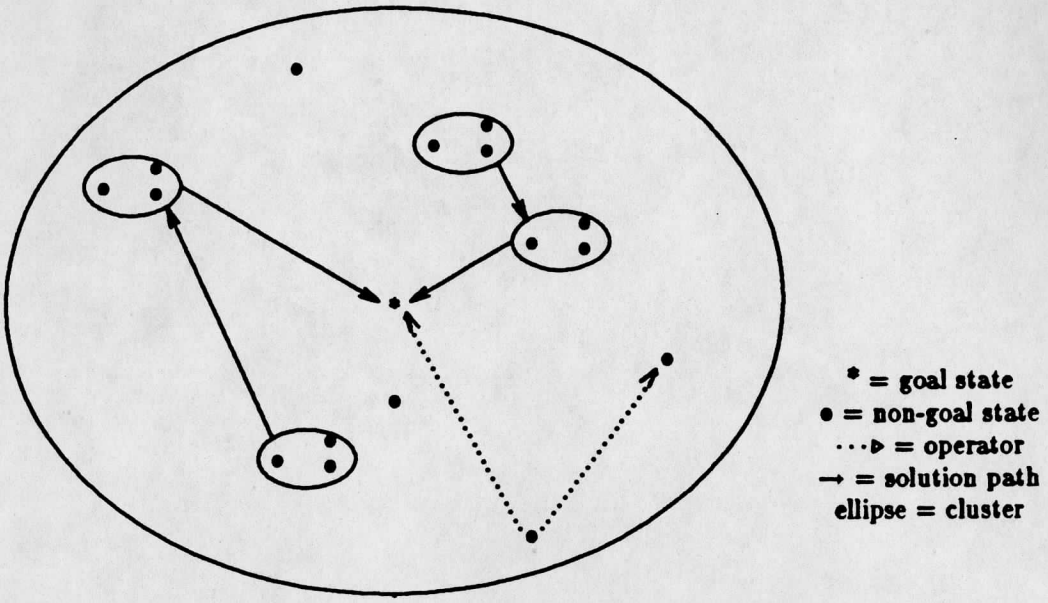


Figure 3

Clusters of States

2) Clusters of States – As shown in figure 3, clusters are groups of states which can be regarded as a single state. This is critical to problem solving because the state space is potentially infinite. For problem solving purposes, clustering replaces an infinite set of states by a finite set of clusters. Clusters are formed by grouping states which play the same role in a solution path. That is, if a solution path transits from $state_1$ to a goal then the group of states in the state space for which the same solution path is effective defines a cluster which includes $state_1$. Learning clusters of states is the subject of chapter 4 which discusses the technique of **perturbation**. Perturbation is an automated method of discovering *when* operators should be applied – i.e., the cluster of states in which each operator is effective.

3) Operator Representations – As shown in figure 4, there is a spectrum of operator representations. At one extreme are *opaque* representations which

Figure 4

A Spectrum of Operator Representations

cannot be easily analyzed. At the other extreme are *transparent* representations which can be analyzed. The world abounds with opaque operators. With opaque representations, the particular state to which the operator is applied and the state resulting from the application can be observed. But the general transformation performed by the operator is hidden. Transparent operator representations, on the other hand, are unnatural in the world. These representations make explicit the transformation performed by the operator. The advantage of transparent representations is that a problem solver can reason with the operator "semantics." Since it is unreasonable to assume that operators are represented transparently, chapter 5 discusses a technique for learning transparent representations from examples of application of opaque operators. The representations learned are called **relational models**. Relational models explicitly represent *what* individual operators do.

This knowledge is useful for solving any problem which can be formulated as a state space search. In general, this knowledge cannot be built into a problem solver because it is difficult to obtain and encode (the knowledge acquisition problem).

This dissertation presents a method for learning this essential state space knowledge. The technique proposed builds on the inter-connection of a problem solver and a learner. The problem solver is proficient at applying operators to states and the learner, observing the problem solver, compiles knowledge which can be used for subsequent problem solving. Initially, the problem solver is presented

with problems without any knowledge of how to proceed. In this naïve state, there are two alternatives: performing uninformed search or asking for help. Since search techniques are not the focus of this research, the latter alternative is adopted. Knowledge is learned when the teacher advises the problem solver of the appropriate action. The problem with the advice is that it is overly specific – i.e., it refers only to the current problem solving state. The learner discovers the general lesson from the specific advice by experimentation. The learner proposes experiments to the problem solver and draws conclusions from the results.

The learning approach presented in this dissertation is general to the class of problem solving tasks which can be formulated with a state space representation. But, to make the ideas more concrete and to compare techniques with other work in learning problem solving, the bulk of this dissertation focuses on applying the general techniques to two specific domains: simultaneous linear equations and symbolic integration.

An Approach to Machine Learning

This section discusses an approach to machine learning by presenting a set of general principles.

The first principle of learning is that it is *incremental*. There are two properties of environments which prevent non-incremental, "wholesale" learning. First, environments are complex. Techniques proved effective on "toy-domains" frequently do not scale up to real-world problems. Second, environments change. This requires an adaptable, intelligent system to maintain problem solving ability in the domain. However, change is gradual and most environmental properties are preserved. A ramification of these observations is that intelligent systems should incrementally incorporate new information about their environment into their relatively stable, complex knowledge base.

A second ramification of the principle of incremental learning is that knowledge should be integrated. New knowledge can be understood and retained only if it relates to existing knowledge. An important issue in learning is this incremental incorporation of new knowledge. The notion of incremental learning is supported by *Martin's Law*: "You can't learn anything unless you almost know it already" [WINS83, p 401].

The second principle of learning is that it should be economical. That is, a learning system must be aware of resource limitations. Learning does not occur in a vacuum but rather accompanies problem solving. Therefore, the need to be economical is more acute with the demands of performance in addition to learning. While there are no measures of the "cognitive economy" of learning systems, there are some examples of the principle. For instance, AM [LENA76] is a program which discovers mathematical concepts by exploring interesting patterns in number theory. AM is highly resource constrained given the enormity of the space of potential concepts and is directed by an agenda mechanism which selects among competing exploration paths. Paths are allocated resources based on their "interestingness." This measure of promise changes dynamically. Paths are abandoned when their exploration exceeds the resource allocation or when alternate paths become more interesting.

Uneconomic learning algorithms make assumptions about the environment which restrict their utility. For example, the non-incremental ID3 algorithm by Quinlan assumes that the set of training instances is static and retained throughout the training session [QUIN83]. As a demonstration of ID3, Quinlan applied the algorithm to learning best moves in chess endgames. He experimented with a training set of 125,000 instances, which was about 9% of the universe. While ID3 generated virtually error-free rules, the assumption that large training sets can be retained throughout the learning process is unrealistic. (Note this does not conclude that learning systems should not deal with massive training sets; they should. In

fact, the enormous quantity of data in real-world learning environments compels us to obey the principle of learning economically.)

The third principle of learning is a bias concerning knowledge representations for learning. There are two broad classes of representation techniques: conceptual (symbolic) and numeric. Conceptual representations are rich, symbolic structures and numeric representations are a compilation of information into a numeric "summary." Samuel [SAMU59][SAMU67] used a numeric representation in a program which learned to play checkers. A static evaluation function was used to select among candidate moves. The function was a weighted sum of features which Samuel believed were relevant to checkers play. The learning task was to adjust the feature weights so that the resulting evaluation function matched the play of the teacher.

In addition to Samuel's success, proponents of numeric representations point out that all human behavior (including learning) is representable by neural networks. The problem with numeric representations, however, is that the mapping from high-level cognitive processes to neural nets spans many levels of representation which are little understood. Until this mapping is understood, symbolic representations are more likely to enable progress in machine learning.

The issue of numeric versus symbolic representations was addressed by the MYCIN project. MYCIN, developed by Shortliffe [SHOR76] in 1975, was an expert system for reasoning about medical illnesses from symptoms. Domain knowledge was encoded using a combination of numeric representations and weak symbolic representations. Production rules are a symbolic representation which allow MYCIN to draw inferences from data. A numeric representation is used to express the degree of confidence in each rule and its inferences. Both representations can be considered a form of compiled knowledge. That is, domain knowledge is "distilled" into rules and weights but as a result much knowledge is implicit in MYCIN and cannot be recovered. Clancey [CLAN83], in attempting to "transfer-back" the expert knowledge in MYCIN to medical students, found that the representations used by

MYCIN hide relevant information. Clancey concludes that a rich, symbolic representation which makes domain knowledge explicit is essential for modelling complex cognitive processes.

This dissertation describes a learning system designed with these principles in mind. If a metric were defined for each principle then this work and others could be plotted on multiple scales. Research in machine learning should strive for "high marks" on these measures.

Survey of Dissertation Chapters

Chapter 2 surveys related work in machine learning. The chapter describes "landmark" learning systems which strongly influenced this research. A high-level description of the learning algorithm and the data structures manipulated are presented for each system.

Chapter 3 introduces the design of the PET learning system. The emphasis of the chapter is on the incremental nature of learning as modelled by PET's *episodic learning*. Episodes encode knowledge of *why* operators are useful by recording the sub-goal achieved by each. Episodic learning enables PET to learn useful problem solving sequences of primitive operators.

Chapter 4 discusses PET's approach to learning *when* individual opearators should be applied. *Perturbation* is a technique for automatically discovering clusters of states which play the same role in operator sequences. These clusters are formed by conducting experiments in the task domain. By employing the technique of perturbation, PET reduces the teacher involvement and speeds learning.

Chapter 5 discusses PET's approach to learning *what* individual operators do during problem solving. An approach is presented for learning a *relational model* of the transformation performed by each operator in a problem solving domain.

Relational models also serve as a variant of heuristic rules. This variance improves problem solving by constraining operator applications. This representation is learned from example applications of problem solving operators.

Chapter 6 describes the integration of the state space knowledge described in chapters 3, 4 and 5 into a powerful learning paradigm. Examples of system performance are presented to demonstrate the inter-play between episodes, perturbation and relational models. This integration permits constraint back-propagation to improve the learning rate and descriptor composition to improve the description language.

Chapter 7 concludes the dissertation with a summary of the contributions of this research. The major shortcomings of PET are also discussed with proposed areas of future research.

CHAPTER 2
Related Work in Machine Learning

Other research projects relate to this thesis in two ways. First, some related work is significant for its influence on the entire field of machine learning. These landmark projects motivate new directions and are discussed because they provide history and context. Second, some related work is significant for its direct applicability to this thesis. These projects are important to this research because they address specific problems in the current paradigm of machine learning. As such they constitute the *normal science* [KUHN70] of machine learning.

Noting this dichotomy of important related work, this chapter addresses the landmark research projects which influence all of machine learning. On the other hand, projects which specifically relate to this thesis are discussed in subsequent chapters. The details of these projects can then be related directly to the learning approach presented in this thesis.

Five Landmark Projects

This chapter surveys five machine learning projects. The purpose of this survey is to provide history and context to the new research presented in this thesis. Each project is described by the goals of the designers. This is made specific with detailed descriptions of the learning algorithm and knowledge representation technique. From this, the contributions and limitations of each learning approach are discussed. This discussion motivates the design of the PET learning system described in chapters 3–7.

The research projects discussed in this chapter are:

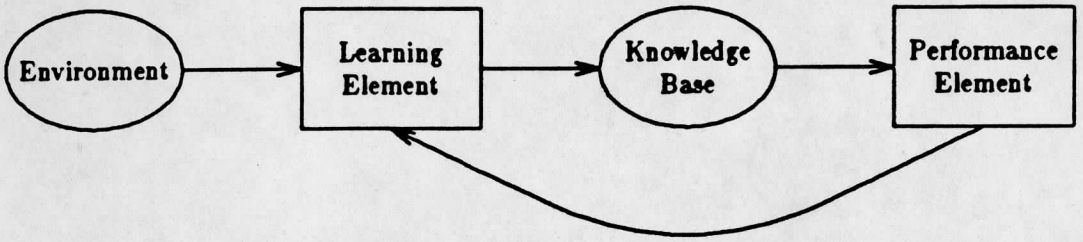


Figure 5

A Model of Learning by Example

1) Samuel's checkers player which learns a state evaluation function that accurately mimics the play of its opponent.
2) Winston's ARCH program which learns by example to identify structured objects.
3) Quinlan's ID3 program which gains efficiency by restricting the description language.
4) Michalski, *et.al.* INDUCE1.2 program which proposes a highly expressive description language and powerful generalization operators.
5) Mitchell, *et.al.* LEX program which addresses the issue of integrating learning with problem solving.

Learning by Example

The five experiments in machine learning discussed in this chapter all demonstrate the technique of *learning by example.* The central issue in learning by example is illustrated in the model shown in figure 5 [COHE82, pp 327–332]. In the learning model, ellipses denote declarative information such as facts presented by the teacher or facts learned from training. Boxes denote procedures. The arrows show the flow of data through the learning system. The environment supplies training to the learning element in the form of examples. The learning element interprets the examples and refines the knowledge base. The performance element

uses the knowledge base to demonstrate its learned ability. Finally, information derived during performance may serve as feedback to the learning element.

The information provided by the environment is too specific. The environment supplies specific examples of general concepts to the learner. The learning task is to generalize this training by separating the relevant details of the examples from the irrelevant details. Since the learning element does not know in advance which details can be ignored, it must form intelligent hypotheses. These hypotheses can be tested by matching them with subsequent training examples. Or, the hypotheses can be encoded in the knowledge base and tested by the performance element.

The performance element provides evidence of the learning system's progress. In its simpliest form, the performance can be evaulated by the teacher to guide subsequent training. An alternative is to use the performance as feedback to the learning element. With this feedback, the learning element can use the performance element as a testbed for validating hypotheses. This is particularly useful for addressing the *credit-assignment problem* [MINS63]. Given a trace of a problem solving task requiring multiple steps, the credit-assignment problem is the problem of assigning credit or blame to individual steps in the solution. This information is useful for subsequent training to avoid bad steps and reward good steps.

This simple model of learning by example provides a framework for discussing the five research projects reviewed in this chapter.

Samuel's Checkers Player

Arthur Samuel's contributions to machine learning are significant. Between 1947 and 1967, Samuel conducted a series of experiments in the domain of checkers playing. His experiments demonstrated such potential that he predicted in 1964 that the problem of machine learning would be solved by now [SAMU64]:

> "At the present time, computers do not learn from their experience. Given a new problem to be solved, no matter how similar it may be to a previously solved problem, we, as humans, must write a new set of instructions, a programme in the jargon of the trade, to specify the solution procedure ... By contrast, when similar tasks are given to a human assistant, he is expected to learn from his experience; and a clerk who has failed to do so is likely to be looking for another position. This problem of machine learning should certainly have been solved well within the next twenty years, and the computer will then become a very much more useful device."

Review of Checkers

Samuel's checkers player [SAMU67] follows the model of learning by example described above. The environment is a teacher who presents board positions to the learning element. Accompaning each board position is advice on the best move to make from the set of possible moves. Equivalently, this advice can be the best board position achievable from the current state. The learner assumes that the teacher has an "expert" board evaluation function. This function measures the quality of the candidate board positions. The teacher selects the best board position from the set of possible positions by applying the function to each. The board position with the highest score is selected.

The learning task is to discover the board evaluation function used by the teacher. This function is assumed to be represented by a weighted feature vector. The goal of the learning is to discover a set of weights, such that the resulting weighted feature vector mimics the state evaluation function used by the teacher. In this sense, the checkers player uses a numerical representation, as described in chapter 1.

The checkers player is provided with a set of 38 features which are potentially relevant to evaluating a board position. Initially, the knowledge base consists of the state evaluation function with all features equally weighted. The performance element selects among possible moves by conducting a limited *alpha-beta* search. The evaluation function is applied to leaf nodes and the best move is backed up.

The teacher provides direct feedback by confirming the move selection or pointing out a better alternative. The learning element then adjusts feature weights to produce performance on this example which matches the teacher's advice.

Before delving into the details of the learning algorithm, it is noteworthy that the teacher's role is automated in Samuel's checkers player. This is done by using the same knowledge base (state evaluation function) as the learning system's performance element. The key difference is that the teacher performs a deeper mini-max search. The teacher's selected move is therefore more informed, since the terminal nodes of a deep search are a better indicator of ultimate success or failure than are terminal nodes of a shallow search. The learner adjusts feature weights such that the limited search yields the same recommended move as the teacher's extensive search.

Samuel's algorithm for inducing feature weights is described with the following algorithm:

GIVEN a set of features $\{f_1, f_2, \ldots, f_n\}$, and corresponding
  feature weights $\{w_1, w_2, \ldots, w_i\}$
INITIALIZE all feature weights to 1
Board ← initial position
LOOP
  $S \leftarrow$ estimate of the quality of Board computed by a
    shallow minimax search using the static evaluation function
    $\sum_i w_i f_i$.
  $T \leftarrow$ the teacher's evaluation of Board computed by a
    deep search with the same evaluation function as above.
  $\Delta \leftarrow T - S$
  IF $\Delta < 0$ THEN S overestimates the quality of Board, so reduce
    weights of features which scored high in S
  IF $\Delta > 0$ THEN S underestimates the quality of Board, so increase
    weights of features which scored high in T
WHILE $\Delta \neq 0$
REPEAT

The goal of the algorithm is to discover a set of weights which mimic the state evaluation computed by the teacher's extensive search. This is viewed as a hill-climbing search through the space of candidate numerical weights. Each move

serves as a training instance which is classified by the teacher and incorporated into the learner's knowledge base by refining weights.

Evaluation of Checkers

There are several major shortcomings of numerical representations. First, the set of features must be provided to the learner. While not attempting to discover new features, Samuel did address a weaker form of this problem – feature selection. Feature selection is the task of choosing those features, from a fixed set of features, which are relevant to the evaluation function. Irrelevant features are removed from the computation. Samuel speculated that some features are relevant to the beginning of checkers play, some to mid-game and some to end-game. During the process of adjusting weights, features with low weights are considered irrelevant and features with high weights are considered relevant. Features thereby fall in and out of consideration as the learning algorithm adjusts weights.

A second shortcoming of numerical representations is that interactions between features are difficult to represent. Simply summing weighted features assumes that each feature is independent. In fact, features are typically not independent and their combinations are quite relevant to the "correct" evaluation function. Samuel addresses the issue of interacting features by explicitly computing and representing a value for the evaluation of every combination of features. These values are stored in a *signature table*, which is an n-dimensional array. Each dimension of the array corresponds to a feature, which in the simpliest case is two-valued (feature present or absent). To obtain a board evaluation, the board is evaluated on n features and a unique cell in the signature table is indexed. While signature tables represent interacting features, they are large and difficult to learn.

A third drawback to numerical representations is that the supporting evidence for a conclusion is irrecoverable from the numerical answer. In the domain of checkers playing, for example, an evaluation of a board position might be based

on sub-goals covered by the board (e.g. "achieved control of board center") or potential advantages of the board (e.g. "from here a King can be won in 3 moves"). The reasons supporting a board evaluation are terms of the polynomial evaluation function but are lost in the numerical representation. This has serious ramifications when the knowledge derived from learning is used for anything beyond performance at the original task. For example, the knowledge cannot be used for teaching, explaining, or assisting learning or related tasks. This issue has been addressed by Berliner in QBKG system which explains its moves in backgammon [BERL80].

In summary, Samuel's checkers program demonstrates both the strengths and weaknesses of numerical representations for learning from examples. It shows the power of machine learning with a simple and straightforward learning algorithm. Samuel discovered shortcomings of the approach and implemented solutions. But, the limitations inherent in the representation could not be overcome without a fundamental change in direction.

Winston's ARCH Algorithm

Winston re-ignited interest in machine learning with the ARCH system. Learning research before ARCH was predominately based on numerical representations, such as Checkers and perceptron training [ROSE62]. From limited successes in selected domains, researchers speculated that learning by numerical weight adjustment would apply to building intelligent machines. Speculation ended abruptly when Minsky and Papert [MINS69] proved that there are limits on the potential of perceptron training.

Winston experimented with a richer knowledge representation and demonstrated the power of learning incrementally from examples. ARCH learned to identify structured objects from examples presented by the teacher. In this section we review the philosophy and design of ARCH.

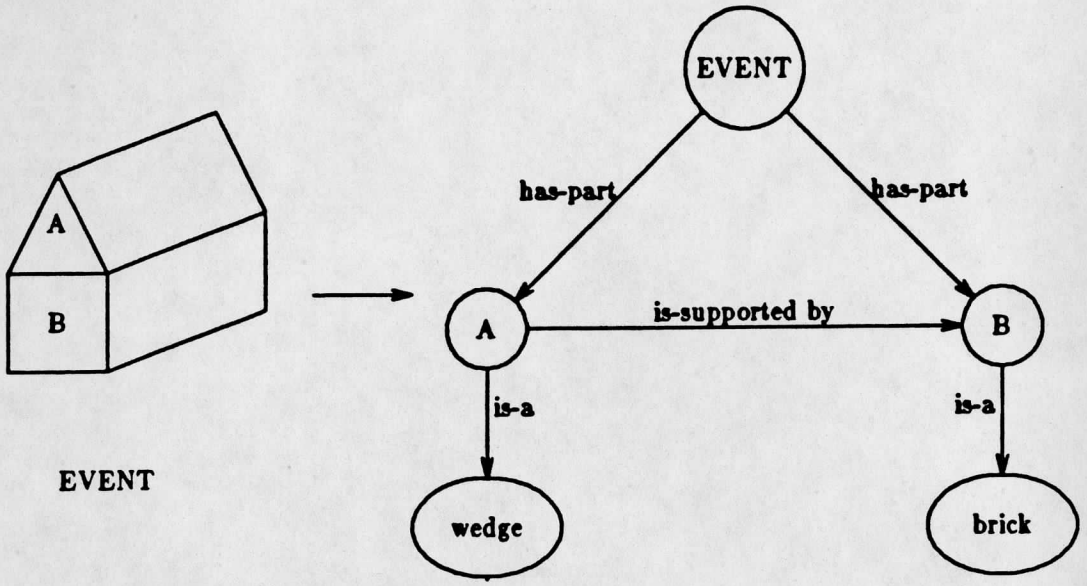


Figure 6

An Example Block Diagram and Semantic Net (from [WINS77, p 31])

Review of ARCH

The learning component of ARCH [WINS75] demonstrates learning descriptions of structured objects from examples. Following the learning model presented above, the learning environment, or teacher, provides ARCH with block diagrams and classifications. These diagrams are particular configurations of structures like arches and bridges. These structures are built from blocks of varying size and shape. The learning element induces a general description of each classification from the specific examples presented by the teacher. With each example, the learning element refines the descriptions in the knowledge base. As these descriptions become more accurate, the performance element is able to correctly classify new block diagrams presented by the teacher.

ARCH makes two significant contributions. The first is the departure from numerical representations. ARCH uses a semantic net representation both for describing examples presented by the teacher and for representing knowledge in the knowledge base. The semantic nets represent blocks (domain objects) and relations

between blocks. Within the network, blocks are represented by nodes and relations are represented with arcs. Arcs which connect two nodes are annotated with binary relations which describe the relationship between the nodes. In addition, nodes are created with unary features which are connected with arcs annotated with HAS-PROPERTY-OF to the objects they describe. Figure 6 is an example of the representation used by ARCH.

An important part of the learning task is to discover the difference between a specific example of a structured object presented by the teacher and the general description of this class of objects stored in the knowledge base. To describe this difference, ARCH builds a third semantic net. Meta-relations (ie. relations between relations) are represented here. For example, if two nets, SN_1 and SN_2, are compared and relation R holds in SN_1 and relation $\neg$ R holds in SN_2, then the relation OPPOSITE can be used to describe the difference.

The second significant contribution of ARCH is the use of *near-misses* in training. The teacher's role is to guide the refinement of concept descriptions stored in the knowledge base (thereby improving the system's performance). ARCH allows the teacher to present to the learning element both positive and negative examples of the concept being taught. Winston observed that negative examples are extremely valuable in correcting concept descriptions which are overly generalized. But, to be useful, a negative example must differ from the current concept description in only a single feature. This enables the learning element to detect the (single) difference between the negative example and the current concept description and make the correct refinement. This restricted class of negative examples are called near-misses.

Given the description of the differences between a training instance and the current concept definition, the algorithm specializes (for negative instances) or generalizes (for positive instances) the concept description. Generalization and specialization operators used are:

1) climbing/descending concept tree - meta-relations and objects are generalized and specialized with concept hierarchy trees. Nodes of the tree are labelled with subsets of meta-relations or objects. If node n_1 is a descendant of node n_2, then n_1 is a subset of n_2 (e.g. a CUBE is a type of BLOCK).

2) dropping conditions - objects considered irrelevant to the concept are dropped by removing them from the semantic net. Note that relationships found irrelevant are simply weakened by (1).

Winston's induction algorithm is described with the following algorithm:

Concept ← first positive training instance
REPEAT
  TI ← a training instance
  $Diff$ ← skeletal-match(*Concept*, TI)
  Concept ← incorporate-diff(*Concept*, $Diff$)
  DISPLAY *Concept*
UNTIL teacher satisfied

FUNCTION skeletal-match(*Concept*, TI)
  skeleton ← match nodes of TI and *Concept* semantic nets
    to find "best" correspondences. Sub-graph isomorphism
    (matching) is NP-complete. However the search is constrained
    by arc annotations. Winston ignores the problem of multiple
    matchings by expecting the teacher to present examples
    examples with little difference. For one approach to handling
    interference matching, see [HAYE78].
  skeletal – match ← annotate *skeleton* with notes describing
    the match. Most common note is INTERSECTION which means that
    both matched nodes in the two graphs point to the same concept
    (node) with the same relationship (arc). The NEGATIVE-
    SATELLITE note means opposite relationships.
    EXIT notes annotate nodes outside of the match. These
    nodes are considered irrelevant to the concept.

FUNCTION incorporate-diff(*Concept*, *Diff*)
incorporate – diff ← heuristically selected "best"
generalization of *Concept* with *Diff*. This is a point of
non-determinism. The algorithm backtracks if the generalization
returned is found to violate subsequent training instances.
Non-determinism arises in the matching process and
the specialization of the concept. Generalizing with
positive instances is not a problem because they loosen
constraints by minimal generalization of relations using a
concept tree. Thus any number of differences can exist
between the current concept and a positive instance (as
long as a skeletal match can be found).
Negative instances are a problem because:
(1) there are multiple features of a training instance
which might account for the "miss"
(2) there are multiple ways to specialize a relation
using the concept tree.
Only "near-misses" are permitted for negative instances.
A near miss differs from the current concept in only one
feature. This constraint mitigates these problems.

Evaluation of ARCH

Winston introduced the use of symbolic knowledge representation for machine learning. This was a useful departure from "traditional" numeric representations and proved particularly useful for representing structured objects. However, simple semantic net representations do have limitations. N-ary relations cannot be directly represented in semantic nets[1]. The arc annotation vocabulary for describing examples permits only conjunctive descriptions (since all relations in the semantic net are assumed to hold simultaneously). The vocabulary for describing general concepts allows weak forms of disjunction and exceptions. Disjunction is introduced via the MAYBE annotation. This is equivalent to saying that a relation (unary or binary) holds OR it does not hold in the description. This is a restricted form of internal disjunction [MICH83] in which the set of permitted values (which are implicit in MAYBE annotation) are TRUE, FALSE, and DON'T KNOW. Exceptions

[1] Although not used by Winston, n-ary relations can be represented with n+1 binary relations. See [DELI79].

are represented with the annotation NOT and MUST-NOT on arcs in the net. The vocabulary for general concepts uses variables to replace constants (e.g. object names) but makes no use of quantification[HEND79].

The remainder of the evaluation of Winston's learning system is characterized as teacher-dependent. Winston assumes that training instances are thoughtfully presented. The teacher must be cognizant of the current state, the goal state, and the internal induction mechanism in order to direct the learning process. By requiring that the teacher know the system's knowledge state and plan the training sequence accordingly, Winston largely avoids certain complications:

1) checking past instances – The induction algorithm performs a depth-first search through the space of concept descriptions. A depth-first algorithm, in order to ensure consistency with past instances, must check past positive instances when specializing and past negative instances when generalizing. Winston does not perform these checks and thus cannot make strong claims of consistent concepts. By assuming that training instances differ from the current concept by only a few features (one feature for negative instances), induction proceeds in small steps. The generalization tends to "stay on track" and minor errors can be corrected with subsequent training instances.

2) interference matching (combinatorial explosion) – For each training instance presented, the induction algorithm finds the "best" match between the instance and the current concept description. The problem of multiple matchings is alleviated by assuming that the graphs being compared are highly similar.

3) guaranteeing maximally-specific generalizations – Positive instances must be presented in the "correct" order if maximally-specific generalizations are to be found. Only one candidate concept description is carried through the induction and generalized and specialized with instances. Permuting the

order of the instances changes the final concept. In particular the concept may not be maximally-specific.

In summary, Winston's ARCH system is significant for its successful use of symbolic knowledge representation at a time when the limitations of numeric representations were casting a dark shawdow on machine learning. Also ARCH demonstrated the role of near-misses in learning by example. However, ARCH can be faulted for over-reliance on the teacher.

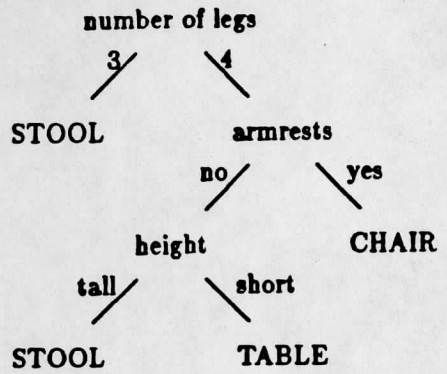
Quinlan's ID3 algorithm

Quinlan's ID3 learning algorithm [QUIN79B, QUIN83] is significant because of its simplicity and efficiency. Referring to the learning model presented above, ID3 employs a simple learning element for acquiring a knowledge base which allows highly efficient performance.

Review of ID3

ID3 demonstrates the trade-off between representational power and efficiency. Quinlan built on Hunt *et.al.*'s experiments in induction [HUNT66]. The environment provides a set of examples to the learning element. Each example in the training set is classified by the concept it belongs to. In the simple case of single concept learning, each example is classified as a positive or negative instance of the concept. The learning task is to induce general concept descriptions which are consistent with the training set. Performance is demonstrated by classifying subsequent examples by applying the learned concept descriptions.

The motivation for ID3 is both efficiently inducing over a large set of training instances and building a concept representation which can classify new instances quickly. ID3 demonstrates that this efficiency can be achieved but representational power is sacrificed. First, the language used for describing examples is restricted to



classes = {chair,stool,table}
features = {number of legs,
armrests present,
height}
values: number of legs={3, 4}
armrests present={*yes*, *no*}
height={*tall*, *short*}

Figure 7

A Sample Decision Tree from ID3

feature vectors. If n features are used to classify instances then the feature vector is of the form:

$$\langle f_1, f_2, \ldots, f_n \rangle s.t. \forall_i 1 \leq i \leq n$$

$$f_i \in \underbrace{\{v_{i1}, v_{i2}, \ldots, v_{im}\}}_{m \text{ possible values for } f_i}$$

This representation is equivalent to a conjunct of features. The features are fixed and are assumed to be independent.

Second, the language used for describing general concepts is a decision tree. This is analogous to a collection of rules each with conjunction and disjunction of feature values. The nodes represent features chosen from the set $\{f_1, f_2, \ldots, f_n\}$ and the arcs from a node labeled f_i are chosen from the set of values for f_i, namely $\{v_{i1}, v_{i2}, \ldots, v_{im}\}$. Leaves of the tree are labelled with names from the set of classes $\{c_1, c_2, \ldots, c_k\}$.

A sample decision tree for three concepts (chair, stool, and table) is shown in figure 7. An instance is classified by the decision tree by traversing the tree from the root. At each internal node labeled f_i, evaluate the instance with respect to f_i and traverse the arc labelled with the result of the evaluation. When a leaf is reached then the instance is classified by the leaf's value. For example, an object

represented with the feature vector ⟨4 legs, no armrests, short⟩ is classified as a table by the decision tree in figure 7.

The goal of the induction algorithm is to minimize the complexity of the resulting decision tree. Several metrics for measuring complexity are described below. Roughly, assume complexity is measured by the average number of arcs traversed in classifying an object using the decision tree. This is minimized by selecting features which are highly predictive of classification to occupy nodes near the root. Also, features which are irrelevant to classification are dropped from the tree entirely. The induction algorithm is surprisingly straightforward and can be implemented using about 20 lines of Prolog (not including the beam search explained below). The PDL of the ID3 algorithm is:

Instances ← a set of training instances
Features ← feature vector $\langle f_1, f_2, \ldots, f_n \rangle$ input from teacher
Domains ← set of domains for *Features* $\{d_1, d_2, \ldots, d_n\}$
  where each d_i is a set of possible values $\{v_{1i}, v_{2i}, \ldots, v_{mi}\}$
Classes ← set of classes $\{c_1, c_2, \ldots, c_k\}$
  Note this set is simply {+,- } for single concept learning.
Rule ← formrule(*Instances*, *Features*, *Domains*, *Classes*)
DISPLAY *Rule*

FUNCTION formrule(*Instances*, *Features*, *Domains*, *Classes*)
  For some *class* ∈ *Classes*
    IF all members of *Instances* fall into *class* THEN RETURN *class*
    ELSE f ← select-feature(*Features*, *Instances*)
    d ← domain from set *Domains* corresponding to f
    RETURN a tree of the form:

```
      f          <-- root labelled f
    / | \        <-- arcs labeled with
   /  |  .\          values from d
                 <-- each child is a subtree
                 created by recursive call:
```

formrule($\{i \mid i \in Instances, eval(i, f) = d_j\}$, *Features*, *Domains*), $1 \leq j \leq m$

The select feature function selects a feature from the set of features which will be used to sub-divide the instances. Several of the more interesting criteria which might be used in the select-feature function are:

1) random selection - while guaranteed to result in a correct decision tree (i.e. complete and consistent with instances given)[2], the rule is sub-optimal. Features may be chosen which do not divide the set of instances into useful subsets. For example, all instances may share one value of the feature, resulting in a node with one descendant. Or, more subtle, the feature may subdivide the instances, but not be criterial to the rule. In this case the subsets have the same mixture of positive and negative instances as the set before subdividing.

2) information theoretic selection - as explained by Quinlan [QUIN83] this method selects the feature (for sub-dividing each node) which, Quinlan conjectures, results in a tree with minimum expected classification time. The feature is selected which is most criterial to the concept being formalized. Criteriality is measured by the ability of a feature to classify instances.

3) minimal cost selection - used in Hunt's original CLS system [HUNT66], this feature selection method balances the cost of evaluating an instance for a feature with the cost of misclassifying the instance. Input to the function are two sets of costs: P_i of measuring the $i+th$ feature of some instance and Q_{jk} of misclassifying it as belonging to class j when it is really a member of class k. The goal is to minimize the combined costs. This method is useful when the cost of evaluating instances is not uniform for all features (e.g. medical diagnosis [QUIN79B]).

When the size of the set of Instances becomes large the goal of efficient rule generation is lost. The problem occurs in two steps of the formrule function: the

[2] As defined in [MICH83], a concept is complete if it covers all past positive instances; A concept is consistent if it does not cover any past negative instances.

first step of determining whether all members of Instances fall into the same class and the last step which involves evaluating each instance with respect to a feature. Due to this practical concern ID3 has been implemented using a beam search. Basically, a subset of the set of instances is selected and a rule is formed which is complete and consistent with respect to the subset. If the rule is not contradicted by any instances in the set of instances, then the beam search terminates. Otherwise, a new selection of instances is made and the search continues. Quinlan [QUIN83] and Michalski [MICH83] have experimented with different criteria for selecting the subset of instances to use at each iteration of the search.

Evaluation of ID3

The expressive power of ID3's description languages is weak. Instances are represented with a feature vector and concepts are represented with a decision tree. The expressive power of a feature vector is minimal. External conjunction is implicit. Disjunction, exception, variables and quantification are not permitted. Decision trees allow both conjunction and disjunction of features. Variables and quantification are not used. This prevents natural encodings of structural and other n-ary relations. Exceptions, also, cannot be represented.

The only generalization rule used in forming the decision tree is the dropping condition rule. This is implicit in the operation of selecting a feature to sub-divide a set of instances. Some features are not used in the final decision tree because they are deemed non-criterial to the "minimal" concept description.

Balancing the weak expressive power of the description language and the modest generalization technique is ID3's efficiency. Efficiency of both the induction process and the resulting decision tree can be measured:

1) the induction algorithm – ID3 was designed to perform induction over large sets of instances. A main source of complexity in the induction algorithm is evaluating an instance with respect to a feature. This operation must

be performed $| F | \times | I |$ times at each node in the tree (where F is the set of features used by I, the set of instances). The efficiency of this step is dependent on the amount of useful knowledge encoded in the feature vector.

2) the decision tree – The final decision tree formed is minimal in that the *expected* classification time of an instance using the tree is no greater than the classification time of an "equivalent" tree. We say two decision trees DT_1 and DT_2 are equivalent if for all instances I, classifiable by either DT_1 or DT_2,

$$classify(I, DT_1) = classify(I, DT_2).$$

This assumes that the set of instances "seen" by the induction algorithm to form a decision tree is representative of the distribution of the larger set of unseen instances.

The efficiency of ID3 is demonstrated with the task of classifying chess endgame positions as won or lost [QUIN83]. In experiments with lost in 2-ply board situations, a decision tree with 83 nodes for 23 features was discovered in less than 3 seconds. The resulting tree classified board positions in .96 seconds, which was about 8 times faster than minimax search.

ID3 requires no teacher assistance during the induction process. All training instances must be provided before processing begins. The induction proceeds depth-first and only one candidate concept description is maintained. Since all instances are present, the problem of checking past instances for consistency (required for incremental learning) is avoided. The rules formed by ID3 are maximally-general in that they do not use features which are not needed to classify instances.

ID3 uses a beam search for efficient learning given a large set of training instances. Given this search strategy ID3 can be made noise tolerant by relaxing the constraint that a decision tree be consistent with all instances. This provides a margin for error. Additional evidence of noise tolerance is given by Quinlan

[QUIN83] when describing the main findings of using a beam search. He found that a correct and consistent tree was formed using only a small fraction of the total set of instances and that the search was not sensitive to the size of the subset selected at each iteration. This indicates that a small margin for error will not disrupt learning in general. A consequence of the ID3 decision tree representation for rules is that the impact of small rule errors on final classification errors is a function of the position of the error in the decision tree. Rule errors near the root of the tree are more serious than errors near the leaves.

In summary, ID3 is motivated by concerns of efficiency. This efficiency is evident both in the induction algorithm and in the resulting decision tree. But there are two shortcomings of the ID3 approach. First, the algorithm is non-incremental. The instances used in training must be presented as a static set to the learning element. Second, the description languages are weak. The instance language is restricted to a feature vector with a fixed set of features and the generalization language is a decision tree.

Michalski *et.al.*'s INDUCE algorithm

The INDUCE algorithm [MICH83] exhibits great inductive power. The main contributions of INDUCE are a multitude of generalization operators and a highly expressive description language. The resulting power is controlled by a heavy influx of domain specific knowledge to guide the induction process.

Referring to the learning model presented above, the INDUCE environment, or teacher, presents the learning element with a set of pre-classified positive and negative examples of a concept. The learning element applies powerful generalization operators (under the guidance of domain knowledge) to form a rule which is consistent with the training set. This rule may be quite terse because of the elegance of the description language.

APC Form	Example	APC↔FOPC
internal conjunction	went(Mary∧Mother,movie)	p(T1∧T2) ↔p(T1)∧p(T2)
internal disjunction	inside(key,(desk1∨desk2))	p(T1∨T2) ↔p(T1)∨p(T2)
relational stmts.	length(rect)>width(rect)	T1 rel T2 ↔rel(T1,T2)
negations	¬(size(ball)=large)	¬(Rel.Stmt.)↔ Rel.Stmt. with opposite rel.
exception (⊣)	went(Mary,movie)⊣ went(Mother,movie)	$S_1 \dashv S_2 \leftrightarrow (\neg S_2 \Rightarrow S_1) \wedge (S_2 \Rightarrow \wedge S_1)$

Table 1

Examples of INDUCE's Description Language

Review of INDUCE

The description language used by INDUCE is built on predicate calculus. Both the instance language and the generalization language allow descriptions of an instance or a concept using Annotated Predicate Calculus (APC). Descriptions are of the form:

$$\underbrace{\langle quantifier\ form \rangle}_{\text{zero or more logical quantifiers}} \quad \overbrace{\langle conjunction\ of\ relational\ statements \rangle}^{\text{predicates in APC}}$$

Michalski defines APC with a set of syntactic additions to first-order predicate calculus (FOPC) and semantic preserving two-way transformations. This allows for more natural encodings of knowledge. Examples of APC usage and mappings to FOPC are given in table 1.

The instance language and the generalization languages differ only in that internal disjunction is prohibited in the former and allowed in the latter. This is a natural restriction since a known instance can be described without this non-determinism. In the generalization language, internal disjunction allows more efficient and natural representation than the traditional external disjunction used in Vere's Thoth system [VERE75], for example.

The generalization rules employed by INDUCE are also very powerful. To some extent this power results from "big step" generalizations. More conservative

generalization rules, such as climbing generalization tree, turning constants to variables and dropping a condition, proceed in small steps and are not as likely to drastically deviate from the "correct" learning path. Michalski introduces the following generalization operators (where ξ stands for an arbitrary expression, $\in$ stands for "is in class", and $\}$ stands for "can be generalized to"):

1) The closing-the-interval rule:

$$\left.\begin{array}{l}\xi \wedge [L = a] \in K \\ \xi \wedge [L = b] \in K\end{array}\right\} \quad \xi \wedge [L = a \ldots b] \in K$$

This rule states that two rules can be generalized if they differ only in the value of a term. The values are assumed to be extremes of a range for which the rule applies.

2) The extension-against rule:

$$\left.\begin{array}{l}\xi_1 \wedge [L = R_1] \in K \\ \xi_2 \wedge [L = R_2] \in \neg K\end{array}\right\} \quad [L \neq R_2] \in K$$

The rule states that given two rules, one positive and one negative for concept K, a generalization is formed which ignores all but one term from each rule. The rule then infers that any instance for which descriptor L does not take value R_2 is positive for class K.

3) Constructive generalization rule:

$$\left.\begin{array}{l}\xi \wedge F_1 \in K \\ F_1 \Rightarrow F_2\end{array}\right\} \quad \xi \wedge F_2 \in K$$

This rule generates an inductive assertion that uses descriptors (in this example F_2) not present in the original instance description. By applying background knowledge and rules which draw conclusions from observed facts, new terms are introduced for use in the generalization language. This allows learned concepts to be used in forming new concepts. This incremental learning rule is also used by Sammut [SAMM81].

Using the description language APC and the powerful generalization rules discussed above, the INDUCE algorithm [MICH83] is summarized by Michalski as:

pos ← set of positive instances from teacher
neg ← set of negative instances from teacher
m ← upper bound on the number of candidate concept descriptions
REPEAT while COLLECTing all values of *rule*
  posinst ← a random selection from *pos*
  rule ← formrule(*posinst*, *pos*, *neg*, *m*)
  reduce *pos* to include only those instances not covered by *rule*
UNTIL disjunction of elements of *rules* covers *pos*
apply collection of FOPC → APC transformations on rules to
  get a simpler expression and DISPLAY

FUNCTION formrule(*posinst*, *pos*, *neg*, *m*)
  candidates ← {*f* | *f* is one conjunct of *posinst*}
  order *candidates*, favoring those which cover the greatest portion of
    pos and reject the greatest portion of *neg*
  expand the set of *candidates* by applying the following inference
  rules to *posinst*:
    1) constructive generalization
    2) problem-specific generalizations defined for the domain
  (this can cover alot of "dirty-tricks" and is not well
  is not well defined by Michalski)
    3) the definitions of previously-learned concepts to determine
  whether parts of *posinst* satisfy some already known
  concepts (again ill-defined).
  order *candidates* using criteria described above.
  delete all but the *m* most preferable descriptions from the set of
    candidates.
  solutions ← {*r* | *r* ∈ *candidates* ∧ *r* is complete and consistent
    with respect to *pos* and *neg*.
  consistent ← {*r* | *r* ∈ *candidates* ∧ ris consistent with
    neg but incomplete with respect to *pos*
  generalize elements of *consistent* by applying:
    1) extension against rule
    2) closing the interval rule
    3) climbing generalization tree rule
  add to the set of solutions those elements of *consistent*
    which are now complete
  formrule ← *m*-best candidates from solutions as ordered
    by above preference criteria

Notice that the formrule function forms a set of maximally general rules which are complete and consistent with the positive and negative instances. This is due to the "seeds" chosen in the first step: maximal generalizations of a selected positive instance. These seeds are then specialized. Since the rules are maximally general, the INDUCE algorithm can form a rule to cover a (non-trivial) subset of the set of positive instances by considering only a single instance. One maximally general rule will then be found which covers the entire set.

Evaluation of INDUCE

The description languages for INDUCE are as expressive as FOPC. Conjunction, disjunction, exception, variables and quantification are permitted. Moreover, the encodings are natural, syntactic modifications to FOPC. In addition, the generalization rules used are powerful. Constructive generalization adds descriptors to a generalization which are not present in the instances used in the generalization.

The INDUCE algorithm is non-incremental. The search technique has components of both a beam search and best-first search. A set of maximally-general candidate concept descriptions is found which is consistent with **all** negative instances and complete w.r.t. **one** positive instance. The best description (as defined by a domain-specific heuristic) is saved. The process repeats with another positive instance. The final rule is the disjunction of best descriptions for some (hopefully small) number of iterations.

INDUCE might demonstrate good noise immunity because of the structure of generalizations found. A generalization is basically a disjunction of terms where each term covers some subset of the positive instances. Noisy instances, assuming they are *very* noisy, are in separate terms which can be removed.

Clearly, the description languages are able to represent any concept for which FOPC is appropriate. But, guided by powerful generalization rules, what class of

concepts can be learned? The problem encountered is the size of the search space. There are three points in the INDUCE algorithm in which non-determinism arises:

1) Expanding the set of candidates by applying inference rules. This involves a non-deterministic selection from three powerful generalization rules with multiple bindings likely.

2) Generalizing consistent concept descriptions in attempt to make them complete also employs three generalization rules.

3) Implicit matching of instances with concept descriptions requires a pattern matcher. This finds the "best" match between an instance and a concept description. It is also used to determine if a concept covers an instance.

Countering this enormous search space, INDUCE employs a body of domain-specific heuristics to select the most promising candidate concept descriptions. These pruning heuristics are essential and are applied at every opportunity during each iteration of the search (*formrule* function performs one iteration). It is surprising that INDUCE does not guide the selection of generalization rules with heuristics. The learning element must select among five different rules to apply, each with the potential of multiple bindings to the current state.

However, as described by Lenat [LENA83], heuristics have a limited domain of applicability. Outside of this domain, a heuristic can be useless or dangerous. We are unable to judge the *heuristic adequacy* [MCCA69] of the INDUCE algorithm. The "knowledge-intensive" approach is proving useful in problem solving, expert systems and natural language processing (among other areas) and should find application in inductive learning as well. However, evaluation of such a system is easily influenced by the quality and quantity of knowledge available to the system. This can conceal the domain-independent, formal properties of an algorithm.

In summary, the INDUCE algorithm contributes an expressive description language and a set of powerful generalization operators under the guidance of a

knowledge intensive learning element. The performance properties of the learning element are difficult to judge because domain specific heuristics are used to prune the search for a generalization.

Mitchell *et.al.*'s LEX Algorithm

There are two major contributions of the LEX system. First, LEX addresses all components of the learning model described above. The result is a system which integrates the environment with learning and problem solving. Second, LEX proposes a representation for heuristics, called *version space*, which aids in the refinement of heuristics with training.

Review of LEX

LEX, described in [MITC78, MITC82, MITC83], is an incremental model for learning problem solving. In a problem solving domain, operators cause state transitions which progress toward the goal. In LEX, the environment, or teacher, presents advice on which operators to apply during problem solving. As with the other learning from examples systems discussed in this chapter, the LEX learning element discovers general rules which are consistent with this teacher training. Given operators $\{op_1, op_2, \ldots, op_n\}$ in a problem solving domain, LEX forms rules

$$\{cond_1 \rightarrow op_1, cond_2 \rightarrow op_2, \ldots, cond_n \rightarrow op_n\}.$$

For each rule, $cond_i$ corresponds to the concept "states in which op_i can be usefully applied."

Domain knowledge provided to LEX is a set of concept hierarchy trees. An example concept hierarchy tree is shown in figure 8. This domain knowledge serves three purposes in LEX. First, nodes of the concept hierarchy trees define the vocabulary of LEX's description languages. Examples presented by the teacher

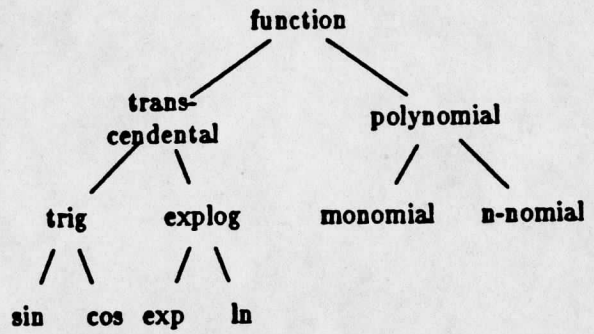


Figure 8

Concept Hierarchy Tree for Functions

must be described using terms in the leaves of the trees. Concept descriptions (generalizations) are described using terms in any node in the trees.

The second use of concept hierarchy trees is to define LEX's generalization operator. Generalization is performed by "climbing the generalization tree." A generalization of an example is formed by replacing a primitive feature of the example with a generalization of the feature. The primitive feature is restricted to be a leaf of a concept hierarchy tree. A generalization of the feature is any (internal node) ancestor of the leaf.

The third, and most significant, use of concept hierarchy trees is to guide LEX's learning element. LEX views concept learning as a search through a space of concept definitions. Both positive and negative training instances are used for navigating through the space. The space of candidates is structured by a partial ordering imposed on candidate concepts. This ordering is defined by a more-specific-than relation between concepts. The ordering defines a tree with the most specific concept at the root and least specific concepts at the leaves. Positive instances force generalizations (i.e. searching deeper in a branch) and negative instances prune branches of the tree.

Traditional breadth first and depth first strategies for searching the space of candidate concept definitions require keeping past instances and verifying that

current candidates do not violate previous instances. LEX avoids these time and space requirements by a search technique analogous to bi-directional search. LEX maintains two sets of candidate concept definitions. One set, S, contains candidates which are equal to or more specific than (as defined by the partial ordering) the correct concept. The other set, G, contains candidates which are equal to or more general than the correct concept. The correct concept lies between these boundaries. Mitchell calls this space between S and G (inclusive) the *version space* of the concept. The version space is implicitly defined and structured by concept hierarchy trees which relate terms in the description languages according to the partial ordering.

The LEX algorithm is described by the following PDL:

Ctrees ← set of concept trees from teacher
posinst ← initial positive instance from teacher
$S \leftarrow \{posinst\}$
$G \leftarrow \{g \mid maximal - generalization(posinst, Ctrees, g)\}$
REPEAT
  $TI \leftarrow$ training instance from teacher
  IF TI is a positive instance THEN
    retain in G only those elements g, s.t. $match(TI, g, Ctrees)$
    for all $s \in S$, s.t. $\neg match(TI, s)$, replace s in S by
      $generalize(s, TI, Ctrees)$
    for all $s \in S$, remove s from S if there exists g
      in G, s.t. $match(s, g, Ctrees)$
    for all distinct pairs of elements s_1 and $s_2 \in S$, remove
      s_2 from S if $match(s_2, s_1, Ctrees)$
  IF TI is a negative instance THEN
    retain in S only those elements s, s.t. $\neg match(TI, s)$
    for all $g \in G$, s.t. $match(TI, g, Ctrees)$, replace g in G by
      $specialize(g, TI, Ctrees)$
    for all $g \in G$, remove g from G IF $\exists s \in S$, s.t.
      $match(g, s, Ctrees)$
    for all distinct pairs of elements g_1 and $g_2 \in G$, remove
      g_1 from G IF $match(g_1, g_2, Ctrees)$
UNTIL $S = G$
DISPLAY S

FUNCTION *generalize*(*concept*, *instance*, *Ctrees*)
RETURN the list of minimal generalizations of *concept* with the (ground) *instance*, w.r.t. *Ctrees*.
Assuming a *concept* definition of n features, the list of generalizations returned will be at most length $n!$.
For each pairing of features in *concept* with features in *instance*, find the minimal generalization by:
generalization ← "null" term for all pairs of
corresponding features $f_i \in$ *instance* and $t_c \in$ *concept*
ctree ← subtree of *Ctrees* s.t. f_i is a node of ctree
APPEND to *generalization* the closest common ancestor of t_i and t_c in *ctree*.

FUNCTION *specialize*(*concept*, *instance*, *Ctrees*)
RETURN the list of minimal specializations of *concept* with the (ground) *instance*, w.r.t. *Ctrees*.
This list is formed by collecting the results of all successful paths through the non-deterministic algorithm:
for all pairs of corresponding features $f_i \in$ *instance* and $f_c \in$ *concept*
ctree ← member of Ctrees s.t. t_i is a node of ctree
replace t_c with a descendant of t_c from ctree in *concept*
IF *match*(*instance*, *concept*, *Ctrees*) THEN
concept ← *specialize*(*concept*, *instance*, *Ctrees*)
EXIT with *concept*

Boolean FUNCTION *match*(*spec*, *gen*, *Ctrees*)
(note: *match* is true iff *spec* is equal to, or more specific than, *gen*)
RETURN *true* IF
for some pairing of features in *spec* with features in *gen*,
for each pair of features $f_s \in$ *spec* and $f_g \in$ *gen*,
there exists *ctree*, a sub-tree of *Ctrees* s.t.
$f_s = f_g$ or
f_s is a descendant of f_g in *ctree*
OTHERWISE RETURN *false*

The LEX induction algorithm terminates when the S and G boundaries meet. This convergence is guaranteed given sufficient (and different) training instances. If the boundaries pass then the training set must be noisy. The ability to estimate confidence in a partially-learned concept (as estimated by the "distance" between S and G) and to determine whether the training set is noisy are significant contributions of the version space representation.

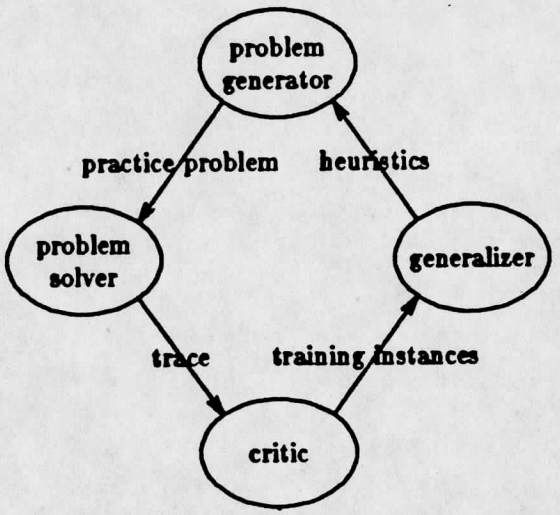


Figure 9

The LEX Learning Cycle (from [MITC83, p 167])

LEX differs from the other induction algorithms surveyed in that it addresses all components of the learning model. Learning becomes a dynamic process which is integrated with problem solving. This dynamic, integrated learning model consists of four components (see figure 9):

1) The problem generator – this component proposes problems to be solved from which LEX can refine its knowledge base.
2) The problem solver – this component attempts to solve the proposed problem using the current knowledge.
3) The critic – this component analyses the resulting search graph. Credit assignment defines "appropriate" applications of a (partially-learned) rule from the knowledge base as those on the solution path. Inappropriate applications are those deviating from the solution path.
4) The generalizer – this component integrates the positive and negative instances of rule application (identified by the critic) with the current knowledge base. This corresponds to the version space induction algorithm.

Evaluation of LEX

LEX sacrifices inductive power for a guarantee of maximally specific, complete and consistent concept descriptions. The concept trees used for generalization and specialization combined with the frontiers of the bi-directional search implicitly define the version space. Also implicitly defined (reconstructable) are two sets of concept descriptions which LEX has ruled out. One set contains those descriptions found inconsistent with some past negative training instance. The second set contains those descriptions found incomplete with respect to some positive instance. LEX proceeds incrementally. For each new training instance, LEX reduces the size of the version space by moving a frontier (effectively moving candidate concept descriptions from the version space to one of the two sets of "ruled out" concepts). This insures that only complete and consistent concept descriptions remain in the version space. Furthermore, the frontiers are moved minimally with each training instance in the sense that only those candidates from the version space which are inconsistent with the current training instance are removed. This guarantees that LEX will find the maximally specific concept description which is complete and consistent with the training set.

The description languages and the generalization rules are central to the LEX version space approach. The description languages are essentially feature lists. The description languages allow natural, domain specific, forms of description (e.g. mathematical notation for the domain of integration problems) but the expressive power is equivalent to a conjunction of features. The only generalization/specialization rule is climbing/descending concept hierarchy trees. By introducing more powerful rules, perhaps operating on a more expressive generalization language, LEX would forfeit the ability to re-construct the search space from the frontiers and to divide the search space into three sets: possible candidates, incomplete candidates, and inconsistent candidates.

One strong advantage of the version space algorithm is that LEX has a dynamic representation of what it knows. This meta-knowledge could be used by LEX's problem generator. The problem generator could measure the degree of convergence on a concept (as measured by the distance between the frontiers of the search graph). A problem could be proposed which bisects the version space, for an optimum learning rate. In fact, the meta-knowledge is not currently used by the problem generator, although Mitchell documents its potential [MITC83]. Another use of this meta-knowledge allows LEX to measure the confidence of a partially-learned concept. This could allow the problem solver and the critic to prudently use partially-learned concepts when expanding a search graph or performing credit assignment. A third use of the meta-knowledge is for learning with noisy training sets. Mitchell and Cohen [MITC78, COHE82] have defined a modified version space algorithm which maintains multiple boundary sets. In the modified algorithm, the sets S_0 and G_0 correspond to the sets S and G in the noise-free algorithm. Added to the storage requirements are S_1-S_n and G_1-G_n where each description in the set S_i is consistent with all but i of the positive training instances and each description in the set G_i is consistent with all but i of the negative instances, for i between 0 and n. If the algorithm detects the crossing of a pair of boundaries S_i and G_i then the algorithm concludes that at least i instances were noisy and looks for convergence on a concept bounded by S_{i+1} and G_{i+1}.

Young *et.al.* [YOUN77] have devised a space-saving modification to the version space algorithm. Basically, they propose eliminating the explicit representation of the frontiers by an implicit representation. Markers are placed at nodes in each concept tree corresponding to the most specific and most general abstractions of the each concept which is consistent with past instances. This is analogous to distributing the frontiers from the concept version space to each of the concept trees involved in the total concept.

In summary, LEX makes two significant contributions to machine learning. First, LEX integrates components of the learning model. This reduces the involvement (influence) of the environment by allowing the learning element to test hypotheses with the performance element. Second, LEX refines heuristics by reducing the version space of each heuristic with incremental training. The version space representation of heuristics is also useful for deriving meta-knowledge on the learning process.

Conclusions

We have surveyed the induction algorithms used by five significant learning systems. These systems are in the class of learning by examples. We have detailed the control mechanism and the knowledge representation used by each. Samuel's checkers player revealed the strengths and weaknesses of a numeric representation for learning. Winston's ARCH system was found to be domain and teacher dependent. By relying on near examples and near misses in the training set, ARCH avoids incorrect concepts and combinatorial explosion. Quinlan's ID3 algorithm buys efficiency and simplicity at the expense of expressive power. Michalski *et.al.* INDUCE program has the expressive power of FOPC, and powerful generalization rules. It represents a significant effort to apply more domain knowledge to the learning process. Finally, Mitchell *et.al.* LEX system guarantees complete and consistent concept descriptions, but employs weak description languages and generalization rules.

This chapter illuminates the commonalities and differences of five significant machine learning algorithms. More important, however, are the ubiquitous tradeoffs of expressive power versus efficiency, domain independence versus strong methods, and teacher guidance versus combinatorial explosion, that characterize the state of the art of machine learning.

CHAPTER 3
Episodic Learning

This chapter discusses a technique for learning operator sequences useful in problem solving. The technique is called **episodic learning**. The goal of episodic learning is to learn useful sequences of operators and heuristics to guide their application. This is important for reducing search for a goal during problem solving. Episodic learning is motivated by the principles of incremental knowledge acquisition discussed in chapter 1.

Defining the Problem

A useful model of problem solving is state-space search as described by Newell and Simon [NEWE72]. Basically, a problem solving task is modelled as search through the space of all possible problem states. One state in the space is selected as the initial state and some set of states are identified as goal states. Operators are defined for the task which transit from one state to another. Starting at the initial state, the problem solving task is to select an operator which, when applied to the initial state, yields a state which is closer to the goal. A sequence of such operator applications transits from the initial state to a goal state. This sequence of operators is called an **episode** [NEWE72, pp 283–303].

Episodes are useful in problem solving because they can be treated as units. Naïve state-space search involves making local decisions at each state in the search of which operator to apply. The strategy used to select operators (the control strategy) can be blind or heuristically guided. In the latter case, heuristics suggest operators which are most likely to advance the search from the current state to a goal state. The heuristics are indexed by features of the current state. While heuristic search

can be an improvement over blind search, heuristically guided search can still be inefficient. This inefficiency is caused by the cost of indexing heuristics and by the fact that local decisions can be faulty. Applying episodes during problem solving mitigates these difficulties by reducing the number of decisions during search.

Newell and Simon observed that people apply episodes during problem solving. They conducted protocal analysis of people solving crypt-arithmetic puzzles. This analysis revealed that human problem-solving activity can be segmented into episodic units. Each episode achieves a part of the problem and advances the search to a recognizable problem sub-goal. Since the episode is a single unit, the problem solver need not labor over multiple local decisions. "[The problem solver] can ignore within-episode detail, and concentrate analysis on the moves from episode to episode" [NEWE72, p 286].

Episodes are effective at reducing non-determinism in state-space search models of problem solving activity. The two issues that this chapter addresses are how useful episodes can be discovered in a problem solving domain and how the application of episodes can be heuristically guided.

Related work on Learning Episodes

This section discusses research projects which address issues in episodic learning. A common theme in this research is that a memory should support episodes as units. Such a memory structure improves indexing of problem solving knowledge. The learning projects discussed here are MACROPS in the STRIPS planning system, rule composition in ACT learning system, and sub-goal learning in the Universal Puzzle Learner.

OPERATOR: pickup(x)

Precondition: ONTABLE(x) ∧ HANDEMPTY ∧CLEAR(x)

Delete List: ONTABLE(x), HANDEMPTY, CLEAR(x)

Add List: HOLDING(x)

Figure 10

An Example STRIPS Operator

MACROPS – The Learning Element of STRIPS

Referring to the model of learning presented in chapter 2, MACROPS is the learning element for the STRIPS performance element. STRIPS is a problem-solving developed by Fikes and Nilsson [FIKE71]. STRIPS forms plans for achieving a pre-specified goal state. The plan constitutes an operator sequence, or episode.

The representation of operators in STRIPS is important to the system. As shown in figure 10, operators are represented by a precondition list, a delete list, and an add list. The precondition list is a formula which must be deducible from the current state description to enable the operator to apply. The delete list is a set of literals which are removed from the current state description by the operator application. Finally, the add list is a set of literals which are added to the current state description by the operator application. In summary, this form of STRIPS operator explicitly lists the requirements for the operator to apply as well as the transformation from the current state to the successor state.

Planning in STRIPS is done with means-ends analysis, similar to that of GPS [NEWE61]. The program selects a *difference* between the goal state and the current state and applies an operator which reduces the difference. The difference consists of a literal of the goal state which is not in the current state description. The operator selected is one which includes the relevant literal in the add list. The selected operator is applied to the current state if its precondition list is satisfied. Otherwise, the unsatisifed preconditions are set as sub-goals to be achieved.

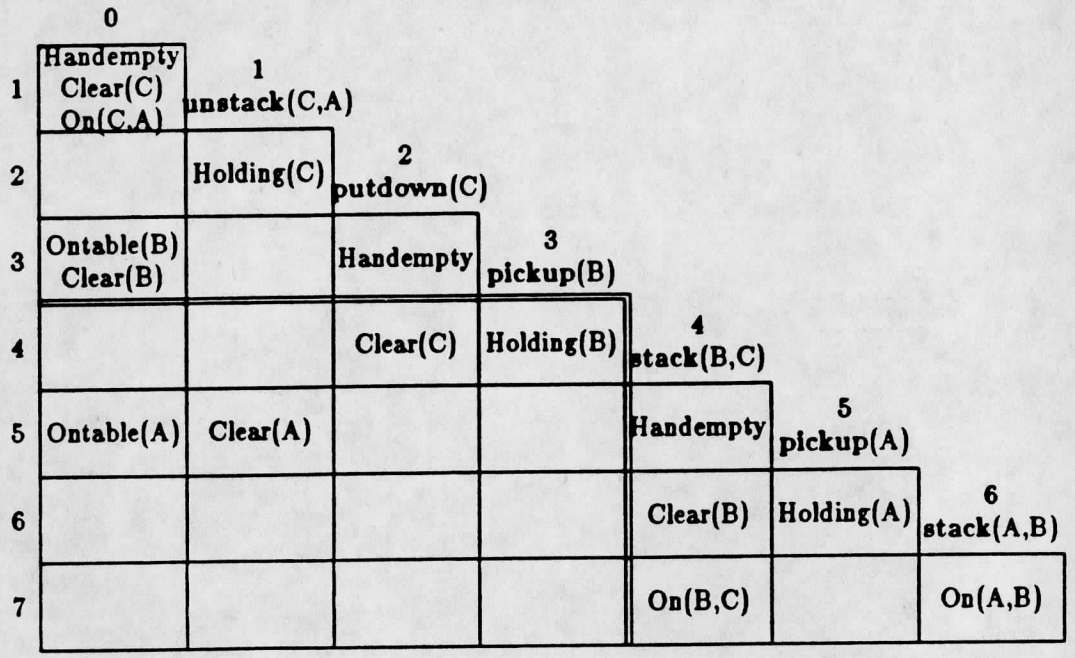


Figure 11

A Triangle Table Representation of a STRIPS Episode

This planning process generates a sequence of operators, or an episode. The STRIPS planning system has no facility for improving performance. The intent of MACROPS [FIKE71] is to remember robot plans that have been generated by STRIPS so that the plan can be reused without regeneration. The plans are stored in *triangle tables* which record the order of application of operators in the plan and how their pre-conditions are satisfied. MACROPS also generalizes plans so they are applicable to a class of problems.

A sample episode from MACROPS is shown in figure 11 (this example and description is from [NILS80, pp 282–287]). A triangle table is a lower diagonal array. The columns are labelled with names of operators applied in the episode. The columns are numbered sequentially from zero, so that the j^{th} column is headed by the j^{th} operator in the sequence. The rows are numbered sequentially from one. If there are n operators in the sequence then there are $n + 1$ rows in the triangle

table. The entries in cell (i, j) of the table, for $j > 0$ and $i < n+1$, are those literals added to the state description by the j^{th} operator that survive as preconditions of the i^{th} operator. The entries in cell $(i, 0)$, for $i < n + 1$ are those literals of the initial state description that survive as preconditions of the i^{th} operator.

Triangle tables explicitly represent the flow of literals through states in a solution path. The solution path is defined by an operator sequence, or episode. In particular, the entries in the row to the left of the i^{th} operator in the episode are precisely the preconditions of the operator. The entries in the column below the i^{th} operator are precisely the literals in the add list of that operator that are needed by subsequent operators or the goal.

The intent of the triangle table representation of episodes is that sub-sequences of the episode can be extracted and re-used without further planning by STRIPS. Such a sub-sequence is called a *kernel.* The 4^{th} kernel is outlined by double lines in figure 11. The entries in this sub-array are precisely the conditions that must be satisifed by a state description in order that the sub-episode of operators 5–6 achieve the goal.

While effective in recording plans, MACROPS has difficulty applying its acquired knowledge [CARB83]. The central problem is that the operators in a MACROPS plan are not segmented into meaningful sequences. Any sequence of operators defines a kernel which can be extracted from the triangle table and re-used as a macro operator. A sequence of length n defines $\frac{n*(n-1)}{2}$ macros. However, few of these sequences are useful. MACROPS offers no assistance in selecting the useful sequences from a plan. An alternative is to forbid the selection of operator sub-sequences and consider the entire plan to be an episode. This results in a large collection of single-purpose macro operators with no branching within the plan. In either case, combinatorial explosion makes planning with the macros impractical. (It should be recognized that MACROPS was designed to control a physical robot, not a simulation. The goal of the MACROPS design was to permit the robot

planner to skip ahead in a plan if the situation allowed or to repeat a step in a plan if the operation failed due to physical difficulties.)

A hierarchical representation of the *purpose* of learned episodes is missing in MACROPS. The triangle table is a flat representation of operators in an episode. One way for MACROPS to overcome the combinatorial explosion of possible episodes derivable from a set of stored plans is to record the purpose of each episode learned. This might include the goal accomplished, the problem solving context, and the strategy and tactics guiding the problem solving. An example of this structuring of problem solving knowledge is Silver's LP learning system [SILV83].

In summary, MACROPS is the learning element for the STRIPS problem solving system. With an explicit representation of operators, MACROPS records each episode in a triangle table. This table captures the flow of literals through the episode. In particular, it allows sub-sequences of operators to be extracted from the episode and re-used as a macro. But, without some guidance for selecting *useful* macros, it is difficult to problem solve with MACROPS episodes due to combinatorial explosion.

Rule Composition in ACT

The ACT learning system [NEVE81, ANDE83] is a theory of learning applied to several domains including that of high school geometry problem solving. ACT is an experiment in *knowledge proceduralization*. The ACT learning element is given a body of declarative knowledge about a problem domain, in this case geometry. The learning task is to convert this knowledge into a procedural form which can be applied by the performance element. This is an example of skill acquisition in that improvement of an existing ability is the concern.

ACT is an example of a learning system in which the results of the performance element feed-back to the learning element (see chapter 2). The performance element

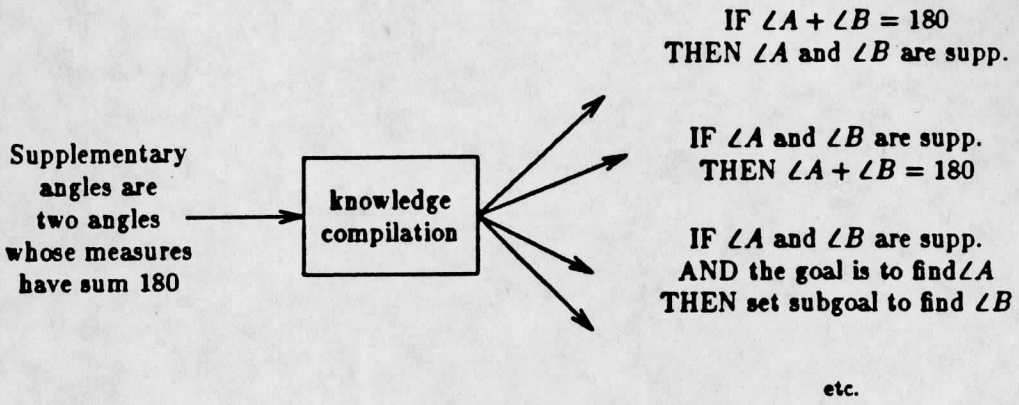


Figure 12

An Example of Knowledge Compilation in ACT

is presented with a geometry problem to solve and a set of general production rules which can apply forwards or backwards. For example, two rules concerning the definition of vertical angles are given by the following two rules:

P1: IF Y is the midpoint of $\overline{XZ}$ THEN $\overline{XY} \simeq \overline{YZ}$ by definition
P2: IF the goal is to prove that Y is the midpoint of $\overline{XZ}$
THEN set as the subgoal to prove that $\overline{XY} \simeq \overline{YZ}$

P1 is a forward working production rule and P2 is a backward working rule. The ACT performance element generates a plan for proving a geometry problem by simultaneously applying forward rules to the "givens" in the problem statement and backward rules to the "to prove" in the problem statement. This bi-directional search creates a proof tree. The problem is solved when the two advancing frontiers of the search meet.

The task of the learning element is to improve the efficiency of creating proof trees. This is done by compiling knowledge from a declarative form to a procedural form. Declarative knowledge, such as the definition of midpoint, is applied to a problem solving task with general interpretive productions. Declarative knowledge is represented in ACT with semantic nets. Procedural knowledge, by contrast, is directly applied to a task because it is encoded in production rules, such as those given above. Anderson draws on the analogy with programming language compilers

which input a declarative statement (a program source) and output a procedural statement (an executable object). An example of knowledge compilation in ACT is shown in figure 12.

The ACT learning element attempts to model the human process of gradually improving problem solving ability with practice. Anderson assumes that people first encode new information declaratively. General interpretive mechanisms then apply this knowledge in multiple circumstances. This performance incrementally improves ability by enabling the learning element to create new procedures (rules) which directly apply this new knowledge without the interpretive step.

There are two learning processes in the ACT system: *composition* and *proceduralization*. Rule composition creates a new production that accomplishes the effect of a series of rules in a single step. This is done by combining both the preconditions and the effects of each rule in the sequence to form a global precondition and effect list for the entire sequence. Rule composition is applied whenever sequences of rules occur in problem solving. This is viewed as a form of episodic learning which improves performance by replacing a sequence of rules with a single rule. This single rule can be used whenever the original sequence applies. Further, the new rule accomplishes the same effect as the sequence.

Rule proceduralization is a learning process which constructs specialized versions of productions. The advantage of proceduralization is that access to memory is reduced during the application of a production rule. This is done by identifying general clauses in the rule which are pattern matched with specific clauses in the state description. By replacing the general clauses by the specific clauses, a production rule is created which applies directly to this situation without the overhead of pattern matching. The penalty, of course, is that many specific rules are created.

The goal of both rule composition and proceduralization is to create production rules which improve the efficiency of problem solving. With composition, this is done by combining a sequence of rules into a single rule with the same effect.

With proceduralization, efficiency is improved by creating rules which reduce access to memory by replacing general clauses with specific clauses found in problem states.

The problem is that this learning approach results in a proliferation of production rules. Composition is applied on every pair of productions that fire in succession. A composition of two rules creates a rule which, if used in subsequent problem solving, is a candidate for further composition. Limiting rule composition is the fact that as rules get "large," they apply to fewer new problem states, thereby reducing further composition opportunities. Also, proceduralization is applied whenever possible. This means that specific rules are quickly produced from general rules which apply to limited situations.

The rules created by ACT's learning element are *strong* procedures [ANZA78]. They improve problem solving by compiling knowledge into a form which can be applied without search. Moreover, the strong, more specific rules which are learned are simply added to the knowledge base. Therefore, ACT can resort to weaker, more general rules when highly proceduralized rules do not apply to a new problem.

But, there is an implicit assumption in ACT that the expense of finding a rule to apply to a problem state is negligible. Since this assumption is unwarranted, it is important to reduce the number of rules under consideration. Anderson calls this problem *tuning the search for a proof.* He proposes four (unimplemented) techniques which reduce the number of candidate rules for each problem state. An alternative is to filter the number of rules which knowledge compilation creates. Learning might be restricted to those rules which satisfy some high-level constraint. For example, rule composition might be applied only to rule sequences which recur or to sequences which achieve a generally useful sub-goal. From both a cognitive and a practical standpoint, it is unreasonable to assume that learning is unfettered by such constraints.

In summary, the ACT system is motivated by the goal of modelling skill acquisition by people. The assumption is that students are presented with declarative knowledge which is initially applied by a general interpretive mechanism. From this performance, procedural knowledge is created which improves problem solving efficiency. This is done by the twin learning processes of rule composition and proceduralization. But, the procedures formed must be useful to general problem solving tasks which recur in the domain. Otherwise, the problem solving search space expands by the addition of new rules without commensurate benefit.

Operator Sequences in UPL

This section discusses the Universal Puzzle Learner (version 2) by Stellan Ohlsson [OHLS82]. UPL consists of a performance element and a three-part learning element. The performance element searches a state space defined by the operators in a problem solving domain. Knowledge of the search space acquired by the learning element guides the search for a goal. If there is no relevant knowledge, then the learning element performs unselective search. Therefore, the performance element can be used independent of the learning element.

The goal of the learning element is to acquire heuristics for improving the performance efficiency. The learning element consists of three components:

1) Good Step Identifier (GSI) – proposes rules which suggest an operator to be applied in a particular situation. A good step is defined to be a transition from the current state to either a goal state or a state which is closer to a goal state.
2) Bad Step Avoider (BSA) – proposes critics which reject an operator in a particular situation. A bad step is defined to be a transition which causes a loop, leads into a dead-end, or leads further away from the goal.
3) Goal Analyzer (GA) – creates goal-setting rules. The purpose of GA is to identify subgoals, which are defined to be useful stepping stones on the

solution path of some top goal. A state in the search space is classified as a sub-goal by GA if it provides opportunity for progress towards the goal.

The knowledge structure constructed by the three part learning mechanism is a set of production rules. These production rules are created during performance at some task. Suppose the performance element transits from state s_1 to s_2 by applying operator OP while seeking goal G. This transition is classified as good or bad by the learning element. This classification is based on a domain-dependent static state evaluation. If the transition is bad, the BSA creates the rule:

IF the current state is s_1 and the goal is G THEN do not apply OP

If the transition is good, the GSI and GA propose the following pair of rules:

IF the current state is s_1 and the goal is G THEN apply OP

IF the current state is s_1 and the goal is G THEN set as subgoal to reach s_2

There is no doubt that the knowledge acquired by the UPL learning mechanism is adequate for problem solving. The rules proposed by the GSI and GA define an episode. The episode is guaranteed to achieve the goal, if it applies. Suppose UPL learns an episode of length n transitions from state S to goal state G. As described above, there are n rules learned by GSI and n rules learned by GA. If this episode is being used by the performance element, subgoal setting rules proposed by the GA back-up from G to S. All states on the solution path from S to G are marked as subgoals. This path is then followed by the operator proposing rules created by GSI. The episode effectively defines a procedure for achieving G form S with no search required.

There is a strong implicit assumption in UPL that static state evaluations can be defined to guide learning and performance. State evaluations are critical for the learning element. The GSI, BSA and GA components use state evaluations to estimate the quality of states. A transition between two states s_1 and s_2 is good if s_2 is closer to the goal than s_1, as judged by the static evaluations of s_1 and s_2.

This determines whether a proposer rule is created by the GSI or a critic rule is created by the BSA.

There are two inherent problem with static evaluation functions. First, they are inaccurate. Reliance on static evaluation functions can result in the frailities common in hill-climbing search. Hill-climbing is frustrated by the problems of local maxima, plateaus, and ridges in the state space [NILS80, pp 22-24]. Second, static evaluation functions are difficult (sometimes impossible) to define. For example, the function which counts the number of tiles out of place is adequate for some initial configurations of the eight tile puzzle but gets stranded on local maxima for other configurations [NILS80, p 24]. But, the broader issue is that even if a "correct" function could be found by exhaustively studying a particular problem domain, learning mechanisms should not rely on this. Reliance on static evaluation functions by learning mechanisms reduces domain independence and restricts their application to domains for which functions can be found *before* learning takes place.

In summary, the Universal Puzzle Learner is a system for learning while doing. The learning mechanism observes performance at one task and creates production rules which improve performance efficiency on subsequent, related tasks. This knowledge is based on local evaluations of paths selected during performance. Unfortunately, these evaluations are based on static evaluation functions applied to states in the search space. This restricts the utility of UPL to those domains in which computable functions exist and can be defined *a priori*.

Learning Episodes for Problem Solving

The PET approach to learning episodes is now described with the reviews of MACROPS, ACT, and UPL as background. With the aid of a teacher, students learn to solve simultaneous linear equations and symbolic integration problems. This section discusses this learning process with emphasis on the role of the teacher, the student's prior knowledge about math, and the result of the learning. These

general comments are then made more specific with a PDL description of the PET learning cycle, which learns problem solving in these two domains.

What the Teacher Does

The teacher plays the role of the environment in the model of learning introduced in chapter 2. To fill this role, the teacher performs the following two functions:

1) The teacher demonstrates effective operator sequences for solving specific problems in the problem domain. From this the student learns episodes for problem solving in this domain. The student must discover the purpose of each operator in the episode. This information is not made explicit by the teacher but is required for constructing episodes whose usefulness extends beyond the small set of examples worked out by the teacher.
2) The teacher presents specific examples of general concepts to the learning element, or student. From this the student discovers the general concepts by induction over the set of examples. This conforms to the model of learning by example. Advice to apply OP to problem state S is a specific example of the general concept "problem states in which OP is useful." As with (1), this form of generalization also extends the student's knowledge beyond the limited training provided by the teacher.

The first of these learning tasks is the topic of this chapter. The second task is elaborated on in chapter 4. These issues are jointly addressed in the PET system, but it is useful to describe them separately here.

As discussed in chapter 2, there is a spectrum of levels of involvement by the teacher in the learning cycle. The teacher's role in the PET learning cycle is to provide advice on which operator to apply to a particular problem when the PET performance element is "stumped." This advice is followed and PET attempts to understand *why* it is appropriate. If PET can determine the purpose of the

operator in this instance, then PET learns from the exercise. Otherwise, PET must "bear-with" the teacher until something understandable happens. In either case, PET continues with the solution path, requesting help when necessary, until the problem is solved.

What the Student Already Knows

A student learning to solve simultaneous linear equations or symbolic integration problems is assumed to have prior knowledge of certain mathematical principles and procedures. This knowledge is in three categories:

1) knowledge of operators in the domain – The student is assumed to know *how* to apply operators to problems in the domain. This involves pattern matching general operator definitions with specific problems and propagating bindings through the operator definition.
2) knowledge of goals in the domain – The student is assumed to know the general form of a solved problem in the domain. This serves to guide solution paths to the final goal.
3) knowledge of generalization – The student is assumed to be able to generalize rules from examples. This is not required for episodic learning and will be covered in detail in chapter 4.

Items 1 and 2 will be explained with examples in section 3.5.

The Result of the Learning Process

The task of the learning element is to discover knowledge about the problem space which improves problem solving in the space. Episodic learning contributes part of this knowledge: knowledge of *why* individual operators are useful in problem solving. That is, episodic learning is concerned with discovering the purpose of individual operators in operator sequences. The assumption is that knowledge of parts contributes to knowledge of the whole.

Episodic learning consists of acquiring two types of state space knowledge: state evaluations and useful solution paths. State evaluations are a measure of the quality of states in the state space. The quality of a state S is measured by the distance from S to a goal state, where low distances indicate high quality. In PET, this distance measure is called the **score** of a state. In particular, goal states have score zero.

Learning state evaluations is done by incrementally backing-up values from known states to "neighboring" states. This process starts with goal states. States s_1 *and* $s_2 \in S$ are neighbors if there is a single operator, OP, such that the application of OP to s_1 yields s_2. States which are neighbors of goal states are assigned a score of one. Then, states which neighbor states of score one (and are not goal states) are assigned a score of two, and so on.

There are two problems with this straightforward approach to state evaluation. First, the state space is potentially infinite and the set of states which need to be evaluated must be limited. Second, the state space does not provide explicit information on which states are neighbors. These two problems are both addressed by the second type of knowledge discussed above: knowledge of solution paths. Solution paths are states which are passed through by an operator sequence, or episode. Those states which are included in solution paths are exactly the set of states which PET evaluates. Also, solution paths make explicit neighboring states in the state space. So, knowledge of state evaluation is intimately connected with knowledge of solution paths.

PET incrementally builds solution paths backwards from the goal state. That is, a solution path of length n states is defined by an episode (operator sequence) of length $n-1$ operators. Consider the episode E defined by:

$$s_1 \overset{op_1}{-} s_2 \overset{op_2}{-} \cdots \overset{op_{n-1}}{-} s_n$$

where s_n is a goal state. PET is constrained to only learn from a state transition in an episode if the purpose of the transition can be understood from existing knowledge. Initially, PET only knows the state evaluation of goal states, which have a score of zero. Therefore, initially PET is restricted to learning transitions to goal states, in which the purpose of the transition is clear. Assuming PET has not learned any other episodes and E is presented by the teacher, PET learns that state s_{n-1} is a neighbor of s_n. This enables PET to assign a score of one to s_{n-1}. Further, PET learns the solution path of op_{n-1} to transit from s_{n-1} to s_n.

From this initial learning, PET expands its ability to understand the purpose of transitions. Now transitions which achieve state s_{n-1} are recognized as useful. If episode E is again presented by the teacher, PET learns that state s_{n-2} is a neighbor of s_{n-1}. This enables PET to assign a score of two to s_{n-2}. Further, PET learns the solution path of op_{n-2} to transit from s_{n-2} to s_{n-1}. Thus, PET is incrementally learning E backwards from the goal state.

State space knowledge is encoded in augmented production rules. The form of a production rule for a transition from $state_1$ to $state_2$ via operator op is:

$$score \; - state_1 \rightarrow op$$

where $score$ is the state evaluation for $state_1$. So, the knowledge derived from episode E above is:

$$\begin{aligned} 1 \; &- \; s_{n-1} \rightarrow op_{n-1} \\ 2 \; &- \; s_{n-2} \rightarrow op_{n-2} \\ &\vdots \\ (n-1) \; &- \; s_1 \rightarrow op_1 \end{aligned}$$

As shown in figure 13, episodic learning constructs a lattice structure of solution paths. Each node of the lattice is a learned sub-goal. Arcs between nodes represent transitions which have proved useful in past problem solving. This knowledge of episodes is applied to new problem solving by navigating through the lattice starting with the initial state. If there are multiple successors of a node, then

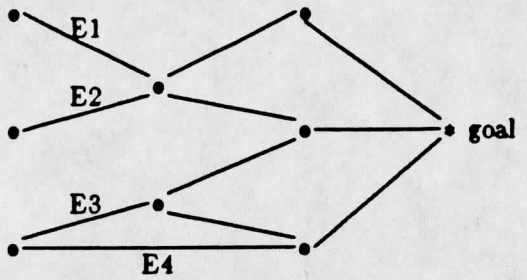


Figure 13

A Lattice of Solution Paths

the problem solver selects the successor with the lowest score. Should the lattice not include the initial state, then weak problem solving methods are applied until a state is reached which is in the lattice. Problem solving with learned episodes replaces search with knowledge of past experience encoded in the lattice.

The PET Episodic Learning Cycle

This section presents the PET learning cycle for episodic learning. A cycle of the learning process begins with a training instance. This instance may either be presented by the teacher or be a state in the solution path of an old problem which PET is still in the process of solving. The second step in the learning cycle is PET's attempt to apply existing knowledge to solve the problem. If PET has already learned an episode which applies to this problem then the episode is applied and this cycle is concluded. Otherwise, the teacher is asked for advice. The final step in the learning cycle is to discover the purpose of the operator suggested by the teacher. If the purpose of the advice can be understood then PET forms an heuristic rule which records this operator's role in the evolving episode. Otherwise, PET does not learn from this training instance.

The PET learning cycle is formally described by the following PDL:

GIVEN an initially empty *rulebase* of heuristics
REPEAT
  get *problem* from teacher
  REPEAT
    IF some *rule* $\in$ *rulebase* matches *problem* THEN
      apply – *episode*(*rulebase*, *rule*, *problem*) (no learning)
    ELSE
      get *operator* advice from teacher
      IF *understand* – *advice*(*rulebase*, *problem*, *operator*, *newrule*) THEN
        learn – *rule*(*rulebase*, *newrule*)
      ELSE no learning
  UNTIL *problem* solved
  DISPLAY *rulebase* for teacher
UNTIL teacher satisfied

Subroutine *apply* – *episode*(*rulebase*, *rule*, *problem*)
$S \leftarrow$ score of *rule*
newproblem $\leftarrow$ APPLY(*rule*, *problem*) (apply rule at head of episode)
LOOP (apply rules in remainder of episode to achieve goal)
WHILE $S > 0$
  DECREMENT S
  SELECT *rule* $\in$ *rulebase* with score S which matches *newproblem*
  newproblem $\leftarrow$ APPLY(*rule*, *newproblem*)
REPEAT
LOOP (apply all possible immediately simplifying operators)
  SELECT *rule* $\in$ *rulebase* with score 0 which matches *newproblem*
  newproblem $\leftarrow$ APPLY(*rule*, *newproblem*)
WHILE *rule* $\neq \emptyset$
REPEAT

Boolean Function *understand* – *advice*(*rulebase*, *problem*, *operator*, *newrule*)
understand – *advice* $\leftarrow$ *TRUE*
IF *operator* simplifies *problem* (goal condition) THEN
  newrule $\leftarrow$ (*problem* $\rightarrow$ *operator*) with score 0
ELSEIF APPLY(*operator*, *problem*) yields a state S which
  enables $R \in$ *rulebase*
THEN *newrule* $\leftarrow$ (*problem* $\rightarrow$ *operator*) with score of *score*(R) + 1
ELSE *understand* – *advice* $\leftarrow$ *FALSE*

Operator	Semantics
combinex(Eq)	Combine x-terms in equation Eq.
combiney(Eq)	Combine y-terms in equation Eq.
combinec(Eq)	Combine constant terms in equation Eq.
deletezero(Eq)	Delete term with 0 coefficient or 0 constant from equation Eq.
sub(Eq1,Eq2)	Replace Eq2 by the result of subtracting Eq1 from Eq2
add(Eq1,Eq2)	Replace Eq2 by the result of adding Eq1 to Eq2
mult(Eq,N)	Replace Eq by the result of multiplying Eq by N

Table 2

Operators in the Domain of Simultaneous Linear Equations

Subroutine *learn − rule*(*rulebase*, *newrule*)
ADD *newrule* to *rulebase*
(Note: This function is the main topic of chapter 4. Specifically, chapter 4 discusses how new knowledge can be integrated into existing knowledge. This integration results in the generalization of knowledge. For now, assume that new knowledge is simply appended to old knowledge.)

Examples of Episodic Learning

This section demonstrates the technique of learning episodes in the PET system. The important points of episodic learning are that episodes grow incrementally backwards from the goal and that episodes encode knowledge of state evaluations and solution paths. Furthermore, episodes are constructed without entire solution paths because learning is based on local decisions. Episodic learning is demonstrated with examples from both the domains of simultaneous linear equations and symbolic integration. These examples are from the PROLOG implementation of PET.

Simultaneous Linear Equations

This section presents multiple examples of episodic learning by the PET system in the domain of simultaneous linear equations. Operators that PET learns to apply for problem solving in this domain are listed in table 2. PET learns episodes which incorporate these operators. Example problems presented by the teacher are expressed in the **instance language** for the domain. PET converts this representation to an internal form. For example, the teacher presented problem:

$$a : 2x - 5y = -1$$

$$b : 3x + 4y = 10$$

is converted to:

$$\{term(a, 2x), term(a, -5y), term(a, 1),$$

$$term(b, 3x), term(b, 4y), term(b, -10)\}$$

where a and b are labels for the equations in the problem.

PET starts with knowledge of how to apply operators to a problem state. But, there is an empty rule base of knowledge of *when* to apply them. Initially, PET has the single goal of simplifying the problem state by reducing the number of terms in the equations. PET builds episodes, or operator sequences, which simplify the problem state.

For the first example, PET is presented the training instance by the teacher:

(State 1)

$$a : 6x + 3y = 12$$
$$b : 6x + 4y = 14$$

with the advice to apply operator sub(a, b). PET applies the operator, which yields the state:

(State 2)

$$a : 6x + 3y = 12$$
$$b : 6x - 6x + 4y - 3y = 14 - 12$$

The operator did not simplify the problem state, since the number of terms increased from six to nine. Using the current knowledge base, PET is unable to understand why the operator is useful. No learning takes place, and PET must "bear with" the teacher.

Now the teacher suggests that operator combinex(b) be applied to the current state. PET applies the operator, yielding:

(State 3)
$$a : 6x + 3y = 12$$
$$b : 0x + 4y - 3y = 14 - 12$$

This state is a simplification with the number of terms reduced to eight. PET can now add knowledge to the rulebase based on this episode.

From this episode, PET learns both an evaluation of state 2 and a solution path from state 2 to the goal. The evaluation is based on the distance to the goal, which, in this case is one. The solution path is combinex(b). The heuristic added to the knowledge base is therefore:

$$1 - - \left\{ \overbrace{\begin{array}{l} term(a,6x), term(a,3y), term(a,-12), \\ term(b,6x), term(b,-6x), term(b,4y), \\ term(b,-3y), term(b,-14), term(b,12) \end{array}}^{State\ 2} \right\} \rightarrow combinex(b)$$

Now the teacher suggests the operator sequence combiney(b), combinec(b), and deletezero(b). The operators are applied sequentially to state 3 and the resulting state is:

(State 4)
$$a : 6x + 3y = 12$$
$$b : 1y = 2$$

Each operator achieved a state simplification. Therefore, each transition is assigned the score one. The rules learned by this training are:

$$1 - - \left\{ \begin{array}{l} term(a,6x), term(a,3y), term(a,-12), \\ term(b,0x), term(b,4y), term(b,-3y), \\ term(b,-14), term(b,12) \end{array} \right\} \rightarrow combiney(b)$$

$$1 - - \left\{ \begin{array}{l} term(a,6x), term(a,3y), term(a,-12), \\ term(b,0x), term(b,1y), \\ term(b,-14), term(b,12) \end{array} \right\} \rightarrow combinec(b)$$

$$1 - - \left\{ \begin{array}{l} term(a,6x), term(a,3y), term(a,-12), \\ term(b,0x), term(b,1y), term(b,-2) \end{array} \right\} \rightarrow deletezero(b)$$

With state 2 learned as a sub-goal with an episode which achieves simplification at state 4, the original training instance for the the subtract operator can be understood. If the teacher re-presents the training instance labelled state 1 with the advice to apply the operator sub(a,b), then PET applies the operator yielding state 2. The learned episode then achieves the goal with no further teacher assistance. Therefore PET is able to understand the transition enabled by sub(a,b). The evaluation for state 1 is done by backing up the value of state 2 and recording the distance to the goal. This is based on the local observation that the evaluation of state 1 is one plus the evaluation of state 2; or two. State 1 is recorded as a subgoal in the space of simultaneous linear equation problem solving. From this training, PET adds the following heuristic rule to the rulebase:

$$2 - - \left\{ \begin{array}{l} \overbrace{\begin{array}{l} term(a,6x), term(a,3y), term(a,-12), \\ term(b,6x), term(b,4y), term(b,-14), \end{array}}^{State\ 1} \end{array} \right\} \rightarrow sub(a,b)$$

Now PET can expand its rulebase and increase the complexity of the learned episodes by learning rules for multiply. First the teacher presents the training instance:

(State 5)

$$\begin{array}{l} a: 6x + 3y = 12 \\ b: 3x + 2y = 7 \end{array}$$

PET's current knowledge does not apply, so the teacher provides the advice to apply mult(b,2). PET applies the operator, yielding:

(State 6)
$$a: 6x + 3y = 12$$
$$b: 6x + 4y = 14$$

which is equivalent to the subgoal defined by state 1. Now current knowledge applies. PET applies the episode headed by the rule recommending sub(a,b). The episode is:

$$\langle sub(a,b), combinex(b), combiney(b), combinec(b), deletezero(b) \rangle$$

which achieves a simplification of state 6. Therefore, PET understands the advice to apply mult(b,2) and new knowledge is added to the rulebase. First, state 5 is evaluated to be a distance 3 from the goal (one plus the already learned distance of 2 for state 1). Second, the following heuristic rule is created:

$$3 \; -- \left\{ \overbrace{\begin{array}{l} term(a,6x), term(a,3y), term(a,-12), \\ \\ term(b,3x), term(b,2y), term(b,-7), \end{array}}^{State\ 5} \right\} \quad \rightarrow \quad mult(b,2)$$

PET learns an episode for "cross-multiply" with further training. Cross-multiply requires multiplying both equations in a pair of simultaneous linear equations with appropriate constants such that the resulting equations have equal x or y coefficients. For this training, PET is presented with the training instance:

(State 7)
$$a: 2x + 1y = 4$$
$$b: 3x + 2y = 7$$

Since current knowledge does apply, PET requests advice. The teacher suggests the operator mult(a,3), which yields the learned subgoal defined by state 5. Current knowledge provides an episode for a solution from state 5, so PET understands the purpose of the teacher advice. PET forms the rule:

$$4 \; -- \left\{ \overbrace{\begin{array}{l} term(a,2x), term(a,1y), term(a,-4), \\ \\ term(b,3x), term(b,2y), term(b,-7), \end{array}}^{State\ 7} \right\} \quad \rightarrow \quad mult(a,3)$$

From this training session, PET learns seven heuristic rules for six operators. These episodes eliminate search by controlling problem solving for subsequent problems covered by the learned rules. The issue of generalizing this knowledge to cover a set of similar problems is addressed in chapter 4.

Symbolic Integration

This section presents a short example of episodic learning in the domain of symbolic integration. As before, PET starts with a set of operators and knowledge of how to apply them, but an empty rulebase of heurisitics to control when they are applied. While there are eighteen operators in this domain, for present purposes, assume there are only two:

$$OP1: \int x^n \, dx \rightarrow \frac{x^{n+1}}{n+1} + C$$

$$OP2: \int a \, poly(x) \, dx \rightarrow a \int poly(x) \, dx$$

OP1 integrates a power of the variable of integration, x. OP2 extracts a constant, a, from the expression being integrated.

As before, PET initially has a single goal. In the domain of symbolic integration it is to eliminate the integral. Incremental learning constrains learning to only those operators applied to states which yield a goal state. Suppose the teacher presents the training instance:

(State 1) $$\int 7x^2 \, dx$$

with the advice to apply OP2. PET follows the advice by binding a to 7 and $poly(x)$ to x^2, yielding:

(State 2) $$7 \int x^2 \, dx$$

State 2 is not a goal state, so PET does not learn from the training.

"Bearing with" the teacher, PET is advised to continue with State 2 by applying OP1. This yields:

(State 3) $$\frac{7x^3}{3}$$

which is a goal state. PET backs-up evaluation from state 3 (the goal state) to state 2 and assigns state 2 the score 1. State 2 is recorded as a sub-goal in the problem solving search space and the heuristic rule:

$$1 \; -\overbrace{7\int x^2\,dx}^{State2} \rightarrow OP1$$

is added to the rulebase.

Now, the original training instance (state 1 with advice to apply OP2) can enable learning. Applying OP1 to state 1 yields state 2 which is a recognized subgoal. State 1 is assigned a score of 2 by backing up evaluation from state 2. From this PET learns the subgoal state 1 and the heuristic rule:

$$2 \; -\overbrace{\int 7x^2\,dx}^{State1} \rightarrow OP2$$

It is important to note that the episode learned is "loosely packaged." That is, rules from the rulebase can be applied in any order so long as the scores of the rules in the sequence are non-increasing. This enables branching within episodes when a shorter path can be selected over a longer path (path length measured by state evaluation). At each state in the solution path, an operator is selected which most advances the progress to a goal.

Summary of Experience with Episodic Learning

This section summarizes PET's experience with episodic learning for learning problem solving in the domains of simultaneous linear equations and symbolic integration.

In the domain of simultaneous linear equations, PET formed a lattice of rules for the operators multiply, subtract, add, combine x-terms, combine y-terms, combine constant terms, and delete zero term. The longest episode was seven rule applications from initial state to goal state. The "head" rule in this episode recommended "cross multiply," which requires two successive multiply operations. As this episode demonstrates, a technique for recording the sub-goals achieved by each operator is essential. Without this supporting justification for each step in a learned solution path this episode could not be learned. As discussed in the previous section, the problem is that the intermediate states in the solution path do not appear to be successively approaching the goal.

In the domain of symbolic integration, PET formed episodes which included rules for eighteen operators. The longest episode was eleven rule applications from initial state to goal state. The initial state was $\int \sin^7 x\, dx$ and the solution sequence is shown in figure 14. Again, the intermediate states in the sequence appear to be diverting from a simplified goal state. Episodes bridge these necessary "digressions."

Once learned, PET can re-play an episode in whole or in part. The limiting factor is that the episode is overly-specific. The next chapter discusses a technique for generalizing episodes so that they apply to a class of problems. The technique utilizes the structures built by episodic learning to partially automate the role of the teacher in learning by examples.

Conclusions

Episodes are important to problem solving because they reduce the complexity of the search for a goal. An episode is a sequence of operators which cause a useful transition in the state space. As such, they can be treated as a single unit. The problem solver can ignore the details inside the episode and concentrate on connecting the "big pieces."

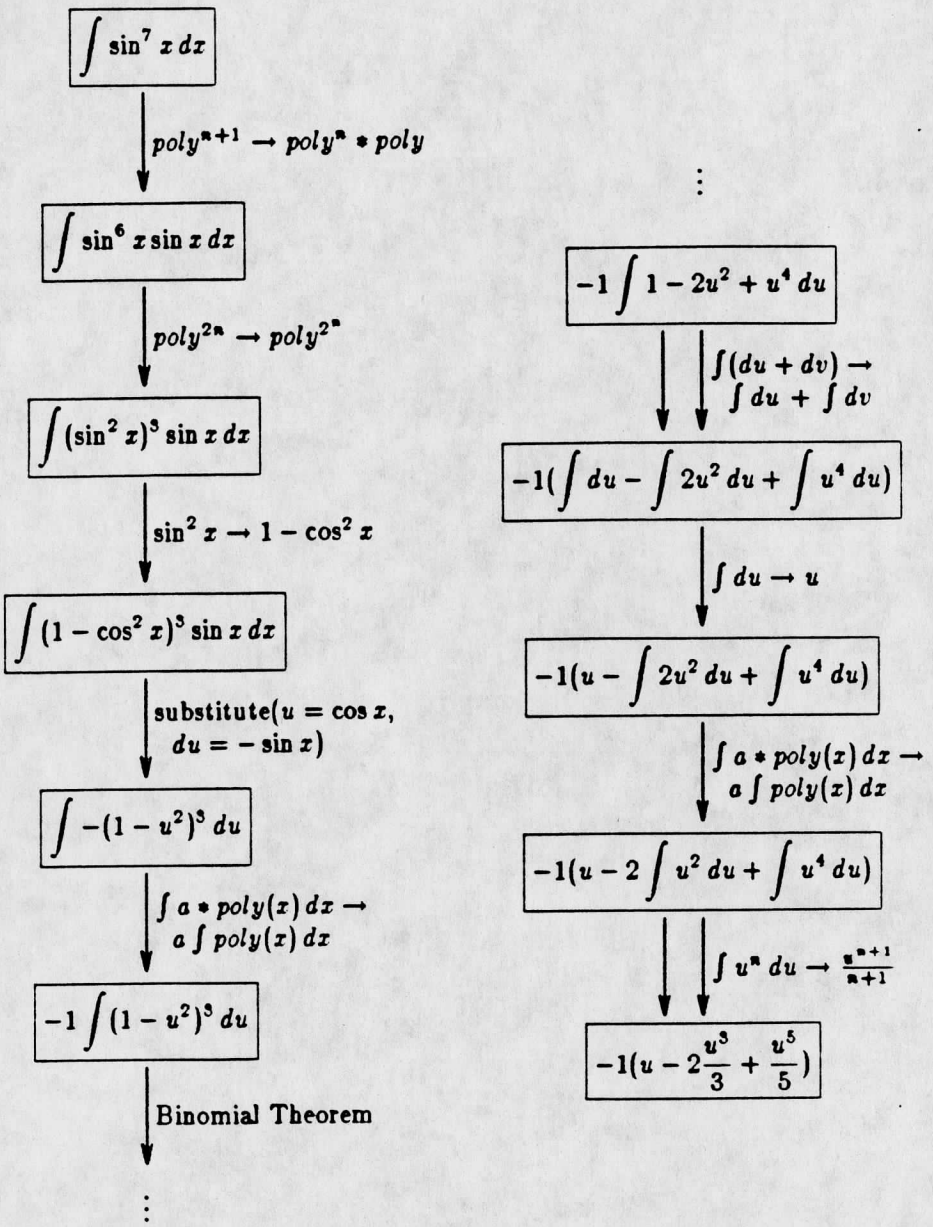


Figure 14

An Episode Linking Eleven Rules

MACROPS, ACT, and UPL are examples of learning systems which discover episodes while problem solving. MACROPS encodes the flow of literals between rules using a triangle table which defines a set of episodes. But, MACROPS does not record the purpose of operators and episodes. This creates a problem with applying MACROPS knowledge because of the combinatorial explosion of candidate

episodes. PET learns state evaluations (scored sub-goals) which support operator sequences by recording the purpose of each operator in the episode. Another viewpoint on this problem is that episodes constructed by MACROPS are treated as individual operators. In PET, episodes effectively replace individual operators as they are constructed.

ACT is a cognitive model of skill acquisition which demonstrates the construction of episodes by composing a sequence of rules into a single procedure. But, ACT lacks controls on learning so that problem solving can be encumbered by a vast corpus of useless rules. PET restricts learning to those episodes which achieve a goal state. The PET system takes into account the problem solving penalty for each new rule added.

UPL demonstrates learning while doing. Performance is improved by adding rules which record episodes which are useful on one problem so they can be replayed on similar problems. The problem with UPL is local decisions are made on state evaluations by *estimating* the distance to a goal. This restricts UPL to those domains for which static evaluation functions can be found. PET avoids this restriction by basing local decisions concerning state evaluations on the *known* distance to a goal. Episodes grow backwards from the goal, so distances are known.

Assumptions concerning the role of the teacher and the student's past knowledge are made explicit in this chapter. The notion of incremental learning of state space knowledge is defined. Episodic learning is shown to involve both discovering state evaluations and operator sequences. This description is formalized with a PDL description of the PET learning cycle. This algorithm will be augmented in chapters 4, 5, and 6 with increased learning capabilities in the PET system.

Episodic learning in the PET system is demonstrated with a set of examples in both the domains of simultaneous linear equations and symbolic integration. These examples follow the PROLOG implementation of the PET system.

CHAPTER 4

Perturbation:

A Technique for Automatic Rule Generalization

In generality there is simplicity. This observation is the motivation for this chapter. The learning element of an intelligent entity must acquire knowledge at the proper level of abstraction. Knowledge which is too general cannot be efficiently applied or may apply in inappropriate situations. Knowledge which is too specific results in an overwhelming mass of detail which conceals the forest with the trees. The "proper" level of abstraction is a balance of these extremes in which knowledge acquisition improves knowledge application.

This chapter discusses a technique for generalizing problem solving knowledge. There are two issues in this discussion. First, how can specific knowledge encoded as rules be generalized such that the knowledge correctly applies to a set of situations? Generalization simplifies the knowledge by removing spurious details which detract from the basic principle. Second, how can the generalization process be automated? Automation simplifies the learning process by removing the teacher from the learning cycle. These issues are first discussed with related work by other researchers and then solidified with an implementation in the PET learning system.

Defining the Problem

This section discusses the problem of generalizing knowledge to an appropriate level of abstraction. This issue is central to many forms of learning but is addressed here with respect to the model of learning by example described in chapter 2. First, some terminology is introduced which will be useful in the discussion.

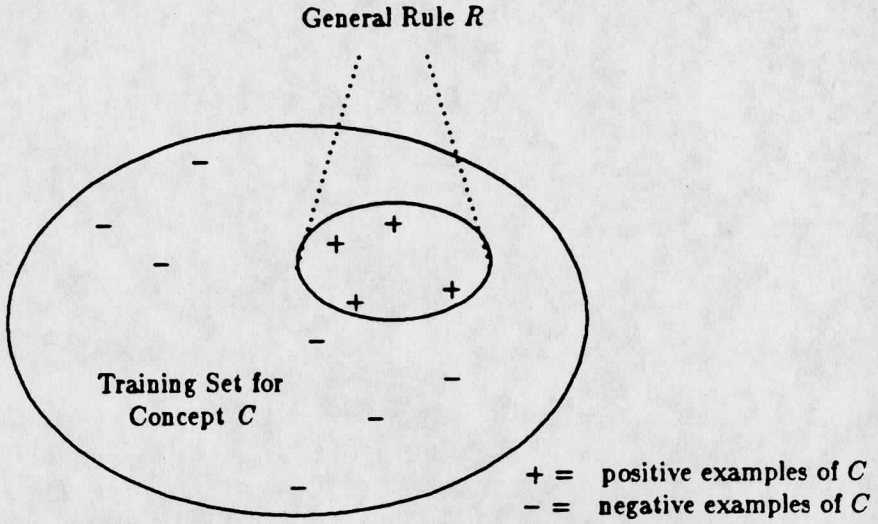


Figure 15

A Generalization from Examples

Terminology for Issues in Generalization

Referring to the model of learning presented in chapter 2, generalizing specific instances is the issue in learning from examples. In this model, the teacher provides training by presenting the learning element with examples of general concepts.

The power of using both positive and negative examples is demonstrated with Winston's ARCH system. As described in chapter 2, the teacher provides ARCH with training instances for a concept which are classified as positive or negative examples. A negative example is one which, for some reason, does not fit the general concept definition. ARCH restricts negative examples to be near-misses which fail to meet the concept definition in a limited number of features (ARCH allows only a single feature mismatch). For example, a tricycle is a near-miss of the general concept of bicycles.

Figure 15 illustrates the generalization of a concept definition from a training set of examples of the concept. From these examples, the learning element abstracts a general rule which is complete and consistent with respect to the examples. That is, a rule R for a concept C is abstracted from a set of examples E such that:

$$E = \{e^+ \mid e^+ \text{ is a positive example of concept } C\} \cup$$
$$\{e^- \mid e^- \text{ is a negative example of concept } C\}$$

$$\forall_{e^+ \in E} R \text{ is a legal generalization of } e^+ \text{ (Completeness Criteria)}$$

$$\forall_{e^- \in E} R \text{ is not a legal generalization of } e^- \text{ (Consistency Criteria)}$$

(A precise definition for "legal generalization" is given below. For now, rely on an intuitive understanding.)

At the core of a learning algorithm is a *description language* which restricts the set of examples and biases the general rule. Language shapes and restricts the class of ideas that are expressible in the language. This observation, called the Whorfian hypothesis [WHOR56], certainly holds true for description languages.

There are two description languages important to machine learning: the instance language and the generalization language. The *instance language* is used for describing examples provided by the teacher to the learning element. The *generalization language* is used for describing general concepts acquired by the learning element. Frequently, the generalization language includes the instance language, yielding a single representation language [MITC82]. This simplifies the generalization task since a representation shift is not required. Furthermore, the simplification is justifiable on the grounds that the generalization language, a superset of the instance language, contains constructs for abstraction which are not required for describing ground instances.

Generalization operators are applied to examples in the instance language to create general concepts in the generalization language. *Generalization operators* "transit" from examples to general concepts by removing specificity from instance language descriptions. Since these descriptions contain explicit details of examples, multiple generalizations are possible. The generalization operators define the legal generalizations that can be applied to examples to create general concepts. For

example, the dropping conditions generalization operator [MICH83] simply removes a specific detail of an example. Using this operator, the example "large, red, square objects" can be generalized to the concept of "large, red objects."

A *legal generalization* G of an example E is any state derivable from a sequence of generalization operators applied to E. The space of legal generalizations is defined by a state space in which E is the initial state and the generalization operators perform state transitions. If a path from E to G, a "goal" state in the space, can be found in this space then G is a legal generalization of E.

This terminology of examples, general concepts, description languages, generalization operators and generalization space is used throughout this chapter.

Generalization Relies on Regularity

As described above, a set of examples of a concept C and the set of generalization operators define a generalization space of candidate definitions of C. With this framework, the generalization step of learning can be viewed as a search of this space. The issue addressed in this chapter is that this space is too large to search naïvely. As described by Mitchell in "Generalization as Search" [MITC78], the learning task is to search this space efficiently.

Why should one have any hope that an efficient algorithm exists for searching the generalization space? Intelligent search is possible because the world is regular. That is, a set of examples of a natural concept in the world share many features in common. The set of examples represent a *prototype* of the concept [MERV81]. Since examples have more features in common than in difference, the function *at least m of n* applied to feature descriptions is useful for defining prototypes [HAMP83].

The AM learning system by Lenat exploites this regularity by guiding learning with heuristics which recommend actions in particular situations [LENA83]. Each heuristic relies on the function:

$$appropriateness(Action, Situation)$$

which measures the usefulness of *Action* in *Situation*. Lenat observes that this is a continuous function of both variables. He postulates two correlaries of this observation:

IF Action A is appropriate in situation S
THEN A is appropriate in situations which are very similar to S

IF Action A is appropriate in situation S
THEN so are most actions which are very similar to A

Lenat also observes that the world is regular in the *Situation* parameter. This regularity makes reasoning by analogy between similar situations a useful problem solving tactic.

It is this regularity and continutity that enables learning of state clusters for problem solving. Viewed abstractly, a state cluster is a group of "neighboring" states in the state space which are basically homogeneous. They constitute a training set for generalization of problem solving knowledge. The state cluster contains positive examples of problem solving concepts. This set of examples can be used to form an initial general concept description. The concepts learned in a problem solving domain are of the form:

The conditions under which operator OP is effective are $\cdots$

(where effectiveness is measured by OP's ability to progress toward a goal – see chapter 3). This concept is defined by a generalization of the set of states in the problem solving state space for which OP is effective. This set of states is a *state cluster* for OP.

In summary, acquiring problem solving knowledge is a form of generalization from a cluster of states. State clusters can be discovered because the world is inherently regular and continuous. Discovering a concept definition which is a legal generalization of a state cluster requires intelligent search of the generalization space. The body of this chapter discusses a technique called perturbation which guides this search.

Related Work on Generalization

This section reviews related work in learning with emphasis on the issue of generalizing concepts from specific examples. Generalization from teacher supplied training instances is relatively well researched. But, there is little research on the issue of automatic generation of instances to enable self-teaching. This section first reviews early research on the theoretical limitations of generalization before discussing recent developments.

The Philosopher's Lament

From a theoretical viewpoint, searching for an efficient induction algorithm is futile. The problem of finding a deterministic finite-state acceptor of minimum size which is compatible with a training set of positive and negative examples is NP-hard [ANGL78, GOLD78]. So, philosopher's may ask, why pursue the impossible?

This cry gained impetus with E. Mark Gold's [GOLD67] theoretical study of language learning. Gold introduced two fundamental concepts: identification in the limit and identification by enumeration.

Identification in the limit views induction as a process which approaches a correct generalization but can never verify this correctness. This is modelled with an induction mechanism M that is provided with an infinite training set T of positive examples of a general concept C. M generates an infinite sequence of conjectures of C from increasingly larger subsets of T. Call this sequence $c_1, c_2, c_3, \ldots$. If there exists some integer n such that c_n is a correct description of C and $c_n = c_{n+1} = c_{n+2} = \ldots$, then M is said to identify C correctly in the limit on this sequence of examples.

M conforms to the model of learning by examples discussed in chapter 2. M incrementally refines its model of C. If, after some finite time, M converges on a correct model of C, then M correctly identifies C in the limit. However, M cannot

determine whether the final model of C is absolutely correct since new, unseen examples may conflict with the model.

Identification by enumeration is a theoretical technique for implementing M. Basically, a domain of concept conjectures is defined which circumscribes candidates of C. All conjectures in the domain are then enumerated, generating descriptions $d_1, d_2, d_3, \ldots$. Given a collection of positive examples of C, identification by enumeration goes down this list to find the first description, say d_i, that is consistent with the examples. d_i is then conjectured as a model of concept C.

However, enumeration is not guaranteed either to achieve correct identification in the limit or to be computable. If the examples provided to M satisfy the following two conditions then identification in the limit is assured:

1) A correct hypothesis is always compatible with the examples given.
2) Any incorrect hypothesis is incompatible with some sufficiently large collection of examples and with all larger collections.

The conditions for an enumeration to be computable are:

1) The enumeration $d_1, d_2, d_3, \ldots$ must be computable from the domain of concept conjectures.
2) It must be possible to compute whether a given concept description is compatible with a given collection of examples.

The problem with these concept identification methods is that they ignore regularities. As argued in the previous section, intelligent induction algorithms exploit regularity in natural domains to form concept prototypes. One way that Gold's theoretical results can be made more practical is to re-organize the space of concept descriptions [ANGL83]. Identification by enumeration uses a simple nonadaptive linear list. The organization of the concept space can be structured to allow the elimination of a set of descriptions when a single incompatibility with

the examples is detected. Furthermore, the structure itself may help select a replacement for a failed hypothesis.

A second form of regularity in natural domains is the categorization of objects in the domain. Rosch and Mervis [ROSC76, MERV81] contend that categories are nonarbitrary. If this contention is valid, then the space of generalizations which must be considered by an induction algorithm is greatly reduced. To demonstrate the nonarbitrary nature of categories, Rosch and Mervis suggest three attributes for classifying animals [MERV81]: *coat* (fur, feathers), *oral opening* (mouth, beak), and *primary mode of locomotion* (flying, on foot). This attribute set defines eight different animal categories. But only two of the eight theoretically possible combinations of attribute values comprise the great majority of existing animal species. By exploiting naturally occuring categories rather than theoretically possible categories, learners significantly reduce the complexity of the learning task.

The next section reviews practical generalization techniques which rely on domain regularities.

Practical Generalization Techniques

This section reviews generalization techniques which have been used in practice. This review is covered in three ways. First, learning systems which acquire knowledge encoded as production rules are briefly reviewed. Second, two approaches to automating the generation of training examples are reviewed. Third, the reader is directed to chapter 2 of this thesis for an in-depth survey of five significant learning systems which perform generalization from examples.

Many learning systems have encoded acquired knowledge with production rules. These systems have proved successful across multiple domains, including poker playing [WATE70], puzzle solving [ANZA78, OHLS82], algebra problems [NEVE78, LANG83, SILV83], arithmetic problems [BRAZ78], and molecular chemistry [BUCH78]. Of these, Neves' system [NEVE78] learns to solve one equation in

one unknown from textbook traces. The system learns both the context (preconditions) of an operator as well as which operator is applied, although the operator has to be known to the system. Silver's system LP [SILV83] carries this research further. LP learns domain knowledge at multiple levels from worked problem solutions. At the lowest level, LP learns rewrite rules by finding differences between consecutive lines in the worked example. At the highest level, LP learns plans by discovering the strategic purpose of each step in the example.

The problem of improving rule learning generalization programs by automating the generation of training instances is little-researched. Two proposals for automatic generation stem from the LEX project. The goal of both projects is to use current knowledge to guide subsequent training.

The first experiment in self-teaching was a theoretical study by Mitchell [MITC78]. Mitchell observed that the version space representation of acquired knowledge (see chapter 2 for a review) contained a meta-level description of the learning process. That is, the learning element can determine its level of certainty in acquired knowledge by examining the version space. Domain knowledge is represented with heuristic rules which guide the application of operators by the problem solver. Each heuristic is defined by a version space of candidate concept descriptions. The version space is bounded by a set of "most specific possible" and a set of "most general possible" heurisitics. All concept descriptions which lie between these two boundaries are compatible with the training examples seen so far.

Mitchell proposes using the version space representation to guide the generation of subsequent training examples in two ways. First, an examination of the version spaces for all of the partially learned heuristics reveals which heuristic is least refined. The heuristic with the largest set of candidate concept descriptions is the one which might most benefit from training. Second, the version space representation is used to generate a training example which most advances the learning

of the selected heuristic. An example is generated which permits the version space of the heuristic to be bisected. The version space is viewed as a set which is well-ordered by the hierarchy of concepts used for generalization. An example which lies equidistant from the two advancing boundaries will exclude half of the concept candidates. Which half is excluded depends on the classification of the example as positive or negative for the concept being learned. While the learning element can derive meta-level knowledge from the version space to guide the generation of training instances, their classification is a problem. More on this below.

The second technique for automating the generation of training instances is implemented in LEX by Mitchell, *et.al.* [MITC83]. This technique generates instances in the same way that perturbation does. Basically, small changes are made to a single training instance to generate a set of highly similar instances. The rational behind this technique is discussed in the next section. One advantage of this generation method over the version space bisection method is computational efficiency.

A shortcoming of both of these training instance generation techniques is that the classification of the generated instance is difficult. Simply asking the teacher for the classification defeats one of the goals: self-teaching. Instead, LEX classifies instances by applying the problem solver to the instance. As described in chapter 2, the LEX problem solver does more than find a solution path for the instance. The problem solver creates a partial search graph which includes failed and abandoned search paths. The critic, another of the LEX system components, analyzes the search graph and classifies state transitions which lie on the shortest solution path to be positive examples and all other transitions to be likely negative examples. Those examples suspected to be negative are confirmed by expanding the search graph to a depth equal to the length of the shortest solution path found thus far. If a shorter path is not discovered on these spurs, then the negative classification is confirmed.

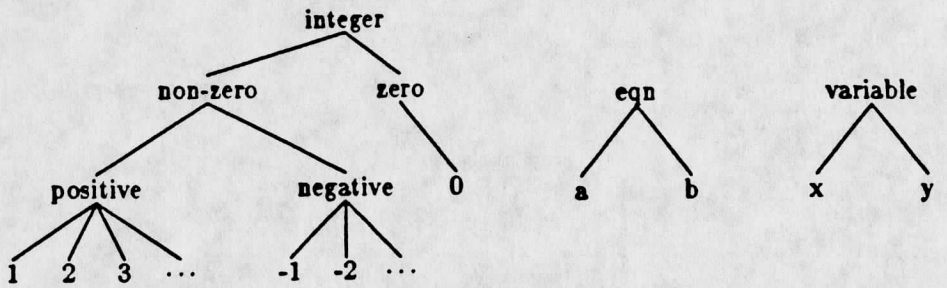


Figure 16

Concept Hierarchy Trees for Simultaneous Linear Equations

Classification is difficult in LEX because there is no explicit learning of episodes. State evaluations are not acquired so sub-goals are not discovered. Only achieving the goal state is recognized as success. The next section discusses the PET technique for self-teaching which makes efficient use of knowledge acquired by episodic learning.

PET Learns General Rules for Problem Solving

PET uses the technique of perturbation to automatically guide the generalization process. While learning concepts for problem solving, perturbation relies on regularities in the domain to form state clusters. Perturbation is a semi-automation of the role of the teacher in learning by examples.

This section discusses PET's algorithm for generalizing knowledge. The first sub-section discusses PET's generalization method. The second sub-section discusses the perturbation method for automatically guiding this process.

PET's Generalization Scheme

PET's generalization scheme is not original. Following Mitchell [MITC78] and Michalski [MICH83], the two generalization operators used by PET are the climbing hierarchy tree operator and the dropping conditions operator. These operators are used in both the domains of simultaneous linear equations and symbolic integration.

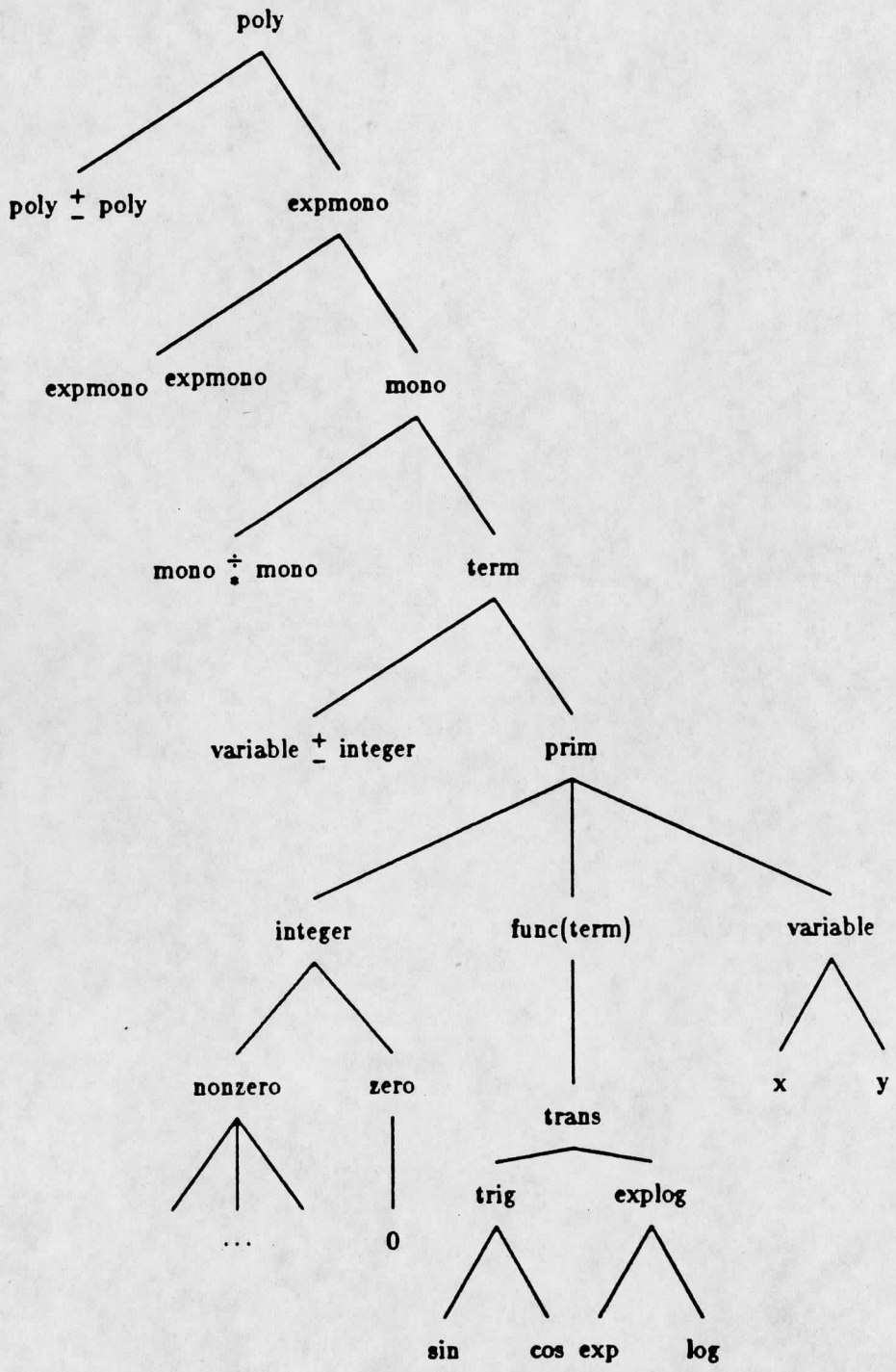


Figure 17

Concept Hierarchy Trees for Symbolic Integration

Figures 16 and 17 show the concept hierarchy trees used by PET for climbing tree generalization. These trees represent an infusion of domain knowledge into the induction process. (Recently, there has been growing interest among researchers to *learn* these hierarchies. Chapter 6 reviews this research and discusses PET's approach to it.)

PET also permits generalization by deleting conditions from an example. Disjunctive generalization is allowed by adding additional concepts (represented as production rules in PET). This covers all of the generalization rules discussed by Michalski [MICH79] except for closed interval generalization.

These few generalization operators applied to the instance language for simultaneous linear equations define an enormous concept space. Consider the single example:

$$a : 2x - 5y = -1$$

$$b : 3x + 4y = 10$$

which is converted to:

$$\{term(a, 2x), term(a, -5y), term(a, 1),$$

$$term(b, 3x), term(b, 4y), term(b, -10)\}$$

The first term, $term(a, 2x)$, has two generalizations of a (a and $eqn(x)$), four generalizations of 2 (2, $positive(N)$, $nonzero(N)$, and $integer(N)$) and two generalizations of x (x and $var(Y)$). The two equations above have four such terms as well as two constant terms, yielding a total of $16^4 * 4^2$ or more than a million possible generalizations! Note that this does not count the additional generalizations that are created by the dropping conditions operator.

During the generalization process, pairs of concept descriptions are matched to find a minimal generalization. This is done with a straight-forward pattern matcher. Some of the terms in each concept description are treated as constants. These are terms which are values of leaf nodes in a concept hierarchy tree. The remaining terms in each concept description are treated as typed variables. These are terms

which have already been generalized from constants to values of internal node in a concept hierarchy tree. The pattern matcher associates terms from one description with those of the other description. The generalizer then finds the minimal common ancestor of the associated terms. If any of the associated terms do not have a common ancestor then a generalization with the current set of associations fails and the pattern matcher tries again.

This simple generalization scheme provides PET with the capability of creating state clusters from examples. The teacher provides the PET learning element with positive and negative examples of a problem solving concept. PET applies the generalization operators to the training set. This defines a generalization space. PET could simply search this space by incrementally refining a concept hypothesis with each new training instance. This approach, adopted by Winston's ARCH system for example (see chapter 2), suffers from over-reliance on the teacher. The next section discusses a technique for reducing this reliance.

Automating Generalization with Perturbation

Perturbation is a technique for reducing the teacher's role in learning by example. The teacher has two responsibilities: generating training instances for a concept and classifying them as positive or negative examples of the concept. From this, the learning element forms a general concept description which is complete and consistent with respect to the training set.

Perturbation relies on inherent regularity in natural domains. Given a training instance I for concept C provided by the teacher, perturbation makes small changes to I. The inherent assumption is that I is prototypical of C. This implies that an instance which is highly similar to I will also be a positive example of concept C. Perturbation generates and classifies the state space "neighbors" of I to form a state cluster of positive examples of C. This state cluster is then provided to a generalization algorithm to form a description of C.

Specifically, perturbation automatically generates a set of positive and negative examples of a concept by a simple two step algorithm. First the examples are generated then they are classified.

Perturbation generates examples by applying a set of perturbation operators to a single teacher supplied example, I. Each perturbation operator selects a single feature f in I and slightly modifies it. Modifications are of two types: replace f by the null feature (effectively removing the feature altogether) or replace f by a sibling of f in a concept hierarchy tree containing f. The latter type of modification generates a set of perturbation operators since f may either have multiple siblings or be in multiple trees. Each perturbation operator generates an example I' which is highly similar to I.

Perturbation classifies examples by exploiting the fact that problem solving domains are *reactive*. Consider a training instance I which is classified by the teacher as a positive example of the concept "states in which operator OP is effective." The generation step of perturbation creates a set of examples which includes example I'. The classification step of perturbation determines whether I' is a positive or negative example of the concept by experimentation. Specifically, I' is a positive example if and only if the *effect* of OP on I' is the same as the effect of OP on I. The effect of an operator on a state is the transition achieved by the operator. So,

$$apply(OP, I) = apply(OP, I') \longleftrightarrow I' \text{ is a positve example}$$

An important advantage of the PET perturbation technique is that examples are efficiently classified. The LEX system [MICH83], by contrast, requires that the problem solver be applied to an example. This involves a full n-level expansion of the search space which terminates when a goal state is reached. If the example lies on the shortest solution path then it is classified as positive, otherwise it is

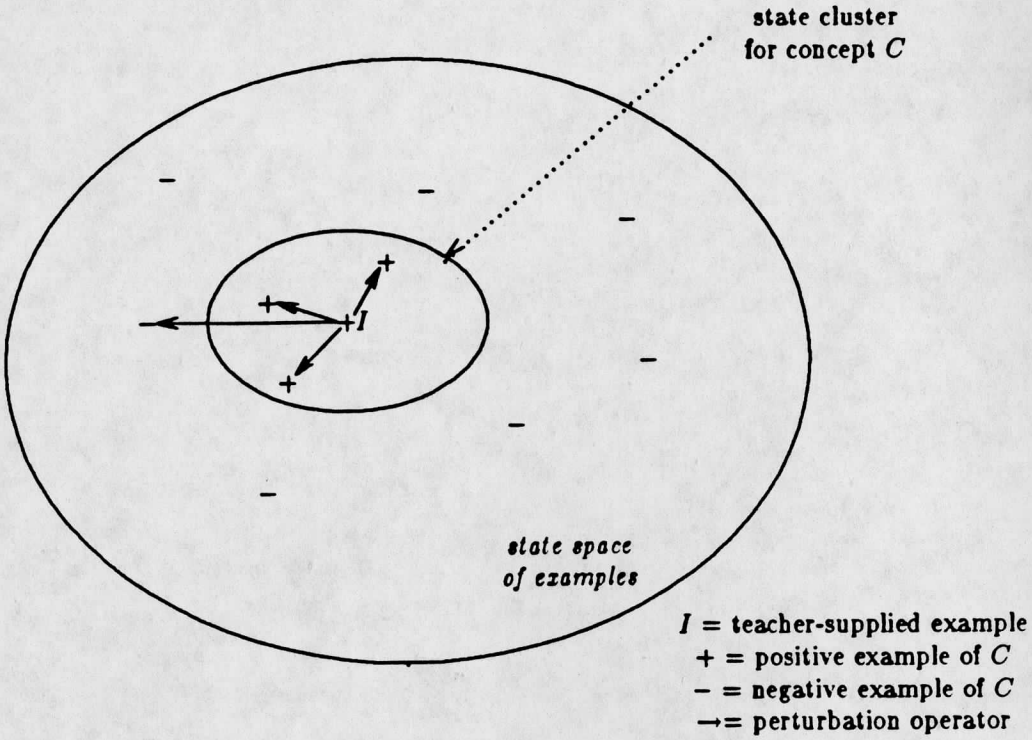


Figure 18

Perturbation Generates and Classifies

Multiple Examples from a Single Positive Example

negative. The perturbation technique simply performs a 1-level search to determine the immediate successor of the example.

Furthermore, examples are efficiently generated by perturbation. Although the efficiency ultimately depends on the instance description language, perturbation simply selects a feature of the instance, finds it in the concept hierarchy tree (a linear search of the leaves, at worst) and selects a sibling (requiring two arc transitions). The perturbation process is shown in figure 18.

The advantage of perturbation is that it removes irrelevant detail from a concept description. The relevance of each feature of an example is tested in two ways. First, the feature is tested to determine if it can be completely removed. Second, if the feature cannot be removed, it is tested to determine if it can be

generalized. Thus, perturbation generalizes each positive example without teacher involvement before generalizing it with the current concept description.

In summary, perturbation is a technique for automatically generating and classifying near-examples and near-misses of a concept. Standard generalization techniques can then be applied. Perturbation semi-automates the learning process by removing spurious details from examples.

The PET Learning Cycle with Perturbation

This section incorporates the technique of perturbation into the PET learning cycle presented in chapter 3. The entire algorithm is repeated here to provide context for the changed code which is boxed.

The PET learning cycle with perturbation is formally described by the following PDL:

```
GIVEN an initially empty rulebase of heuristics
REPEAT
  get problem from teacher
  REPEAT
    IF some rule ∈ rulebase matches problem THEN
      apply − episode(rulebase, rule, problem) (no learning)
    ELSE
      get operator advice from teacher
      IF understand − advice(rulebase, problem, operator, newrule) THEN
        learn − rule(rulebase, newrule)
      ELSE no learning
  UNTIL problem solved
  DISPLAY rulebase for teacher
UNTIL teacher satisfied
```

```
Subroutine apply – episode(rulebase, rule, problem)
S ← score of rule
newproblem ← APPLY(rule, problem) (apply rule at head of episode)
LOOP (apply rules in remainder of episode to achieve goal)
WHILE S > 0
  DECREMENT S
  SELECT rule ∈ rulebase with score S which matches newproblem
  newproblem ← APPLY(rule, newproblem)
REPEAT
LOOP (apply all possible immediately simplifying operators)
  SELECT rule ∈ rulebase with score 0 which matches newproblem
  newproblem ← APPLY(rule, newproblem)
WHILE rule ≠ ∅
REPEAT

Boolean Function understand – advice(rulebase, problem, operator, newrule)
understand – advice ← TRUE
IF operator simplifies problem (goal condition) THEN
  newrule ← (problem → operator) with score 0
ELSEIF APPLY(operator, problem) yields a state S which
  enables R ∈ rulebase
THEN newrule ← (problem → operator) with score of score(R) + 1
ELSE understand – advice ← FALSE
```

```
Subroutine learn − rule(rulebase, newrule)
  Decompose newrule into components:
    state ← LHS(newrule)
    OP ← RHS(newrule)
  Generalize newrule using perturbation:
    currgen ← state
    REPEAT
      SELECT perturbation operator P
      APPLY P to state, yielding state'
      IF apply(OP, state) = apply(OP, state') THEN
        currgen ← minimal generalization of state
          and state'
      UNTIL there are no more perturbation operators
    (now currgen is a description of a state cluster of positive
    examples of the concept "states in which OP is effective.")
  Integrate the new rule into the rulebase:
    genrule ← (currgen → OP) with score of newrule
    IF a member of rulebase can be generalized to cover newrule,
      THEN replace member by generalization
      ELSE add newrule to rulebase
```

Examples of Guiding Generalization with Perturbation

This section demonstrates the technique of perturbation in the PET system. The important points of perturbation are that a set of near-examples and near-misses are automatically generated and classified. These classified examples are then provided to the PET induction algorithm which forms a concept description which is complete and consistent with respect to the training set of examples.

Perturbation is demonstrated with examples from both the domains of simultaneous linear equations and symbolic integration. The examples are from chapter 3 in which overly specific rules are learned by episodic learning. Perturbation guides the generalization of these rules.

Simultaneous Linear Equations

This section presents multiple examples of the use of perturbation to generalize rules acquired by episodic learning. As described in chapter 3, these rules encode heuristic knowledge of operators which are effective in particular states. For example, the first rule learned in chapter 3 recommends the operator combinex(b) in state:

(State 1)
$$\begin{aligned} a &: 6x + 3y = 12 \\ b &: 6x - 6x + 4y - 3y = 14 - 12 \end{aligned}$$

From this, episodic learning forms the rule:

$$1 - - \left\{ \begin{array}{l} \overbrace{term(a, 6x), term(a, 3y), term(a, -12),}^{State\ 1} \\ term(b, 6x), term(b, -6x), term(b, 4y), \\ term(b, -3y), term(b, -14), term(b, 12) \end{array} \right\} \rightarrow combinex(b)$$

The goal of perturbation is to discover a general description of a cluster of states in which combinex(b) is effective. This single positive example provided by the teacher does not adequately restrict the space of candidate concept descriptions. A few of the candidates (selected from the space of millions) are:

$$1 - - \left\{ \begin{array}{l} \overbrace{\boxed{}, term(a, 3y), term(a, -12),}^{State\ 1} \\ term(b, 6x), term(b, -6x), term(b, 4y), \\ term(b, -3y), term(b, -14), term(b, 12) \end{array} \right\} \rightarrow combinex(b)$$

$$1 - - \left\{ \begin{array}{l} \overbrace{term(a, 6x), term(a, 3y), term(a, -12),}^{State\ 1} \\ term(b, 6x), term(b, \boxed{-5}x), term(b, 4y), \\ term(b, -3y), term(b, -14), term(b, 12) \end{array} \right\} \rightarrow combinex(b)$$

$$1 - - \left\{ \overbrace{\begin{array}{l} term(a, 6x), term(a, 3y), term(a, -12), \\ term(b, 6x), term(b, -6x), term(b, 4y), \\ term(b, -3\boxed{x}), term(b, -14), term(b, 12) \end{array}}^{State\ 1} \right\} \rightarrow combinex(b)$$

Perturbation operators are applied to the example in state 1 to generate a set of highly similar examples. Each perturbation operator selects a feature of the example and either deletes it or replaces it by a sibling of the feature found in a concept hierarchy tree. Four of the resulting examples are:

$$\begin{array}{l} a: 6x + \square = 12 \\ b: 6x - 6x + 4y - 3y = 14 - 12 \end{array} \qquad E_1$$

$$\begin{array}{l} a: 6x + 3y = 12 \\ b: \boxed{7}x - 6x + 4y - 3y = 14 - 12 \end{array} \qquad E_2$$

$$\begin{array}{l} a: 6x + 3y = 12 \\ b: 6x - \square + 4y - 3y = 14 - 12 \end{array} \qquad E_3$$

$$\begin{array}{l} a: 6x + 3y = 12 \\ b: 6x - 6x + 4y - 3y = \square - 12 \end{array} \qquad E_4$$

The second step of the perturbation process is to classify each example as positive or negative for the concept being learned. In this case, the concept is "states in which combinex(b) is effective." Effectiveness of combinex(b) is determined for each of the generated examples. The score of the current rule of the operator says that combinex(b) should be one step from the goal. That is, the effect of combinex(b) is that it immediately simplifies the current state by reducing the number of terms.

PET applies this effectiveness criteria to each of the four generated examples above. E_1 is classified as a positive example because combinex(b) is effective at reducing the number of terms in E_1. PET minimally generalizes the state

description of the current rule for combinex with E_1, yielding a new rule:

$$1 - - \left\{ \begin{array}{l} term(a, 6x), \boxed{}, term(a, -12), \\ term(b, 6x), term(b, -6x), term(b, 4y), \\ term(b, -3y), term(b, -14), term(b, 12) \end{array} \right\} \rightarrow combinex(b)$$

The major effect is to delete the condition on the y-term of equation a. This is a spurious detail which is irrelevant to the general concept.

Generated example E_2 is also classified as positive. The minimal generalization of the state description of the current rule with E_2 yields the new rule:

$$1 - - \left\{ \begin{array}{l} term(a, 6x), term(a, -12), \\ term(b, \boxed{positive} * x), term(b, -6x), term(b, 4y), \\ term(b, -3y), term(b, -14), term(b, 12) \end{array} \right\} \rightarrow combinex(b)$$

The major effect of this learning is to recognize that one of the x-coefficients of equation B can be *any* positive integer. Notice that the constraint on this coefficient is still overly constrained since negative integers are excluded. Negative integers are not tested by perturbation in this example because they are not immediate siblings of the original coefficient 6. "Distant" siblings can be tested at the expense of an increase in the number of perturbation operators. Rather that incur this expense, PET relies on the teacher for subsequent training to further refine this rule constraint. This is one of many instances in which PET would benefit from intelligent selection of perturbation operators. Chapter 6 discusses a technique which allows PET to test distant siblings with some assurance that the test will be fruitful.

Generated example E_3 is classified as negative because the combinex(b) operator does not apply. This negative information is not used by the PET induction algorithm. Negative examples are useful for correcting over-generalization in concept definitions. This does not occur in PET since only conservative, minimal generalizations are performed.

Example E_4 is classified as positive. This yields the concept description:

$$1 - - \left\{ \begin{array}{l} term(a, 6x), term(a, -12), \\ term(b, positive * x), term(b, -6x), term(b, 4y), \\ term(b, -3y), \boxed{}, term(b, 12) \end{array} \right\} \quad \rightarrow \quad combinex(b)$$

After the perturbation process the heuristic rule for the operator combinex(b) is highly refined. For the single training instance in state 1, there are thirty perturbation operators. Fifteen of these test if a feature can be removed and fifteen test if a relevant feature can be generalized. The result of minimally generalizing over the generated examples which are classified as positive is the rule:

$$1 - - \{term(b, positive_1 * x), term(b, positive_2 * x)\} \quad \rightarrow \quad combinex(b)$$

where $positive_1$ and $positive_2$ denote two positive integers which are not necessarily equal.

This rule is significantly more refined than its original ancestor above. One rough measure is the number of constraints in the state description. This count is reduced from nine to two by perturbation. Moreover, the two remaining constraints are generalized to cover a set examples. Most importantly, this learning does not involve teacher participation.

The second example of using perturbation to guide the generalization of heurisitic knowledge about linear equations is for the subtract operator. In chapter 3, PET is presented with the following positive example for the operator sub(a,b):

(State 2)
$$\begin{array}{l} a : 6x + 3y = 12 \\ b : 6x + 4y = 14 \end{array}$$

From this training, PET acquires the following rule by episodic learning:

$$2 - - \left\{ \begin{array}{l} \overbrace{term(a, 6x), term(a, 3y), term(a, -12),}^{State\ 2} \\ term(b, 6x), term(b, 4y), term(b, -14), \end{array} \right\} \quad \rightarrow \quad sub(a, b)$$

Again, the rule is overly specific and perturbation is used to remove spurious detail.

In this case there are twenty perturbation operators and each generates an example which is highly similar to state 2. Four of these examples are:

$$\begin{array}{ll} a:\square + 3y = 12 & a:6x + \square = 12 \\ b:6x + 4y = 14 & b:6x + 4y = 14 \\ E_1 & E_2 \end{array}$$

$$\begin{array}{ll} a:6x + 3y = 12 & a:6x + 3y = 12 \\ b:\boxed{7}x + 4y = 14 & b:6x + \square = 14 \\ E_3 & E_4 \end{array}$$

PET classifies each of the instances by determining the effectiveness of the sub(a,b) operator on each. The score of the current rule for sub(a,b) is 2. This means that the operator is effective for a state if its application enables a rule with a score of 1. This ensures that the transition progresses toward the goal.

E_1 and E_3 are classified as negative examples. The operator sub(a,b) is not effective if the x-term of either equation is modified. If the x-terms are modified simultaneously and equally (for instance, each coefficient is changed to a 7) then the operator is effective. However, the size of the set of perturbation operators is kept relatively small by excluding such modifications.

E_2 and E_4 are classified as positive examples. A minimal generalization of the state description of the current rule with E_2 followed by E_4 yields the new rule:

$$2 \; -- \left\{ \begin{array}{l} term(a, 6x), \square, term(a, -12), \\ term(b, 6x), \square, term(b, -14), \end{array} \right\} \rightarrow \; sub(a, b)$$

The effect of this generalization is to remove the constraint that the equations contain y terms.

The heuristic rule for sub(a,b) after the perturbation process is:

$$2 \; -- \; \{term(a, 6x), term(b, 6x)\} \rightarrow \; sub(a, b)$$

While this is a significant improvement over the original rule, further refinement requires another training example from the teacher. If the teacher provides the positive example:

$$a: -2x + 3y = 4$$
$$b: -2x + 4y = 6$$

then PET forms the minimal generalization:

$$2 \; -- \; \{term(a, nonzero_1 * x), term(b, nonzero_1 * x)\} \quad \rightarrow \quad sub(a, b)$$

In this rule, the x-coefficients are forced to be equal since they must bind to the same variable ($nonzero_1$). Thus, PET converges on the correct rule for sub(a,b) with only two training instances provided by the teacher.

Note that learning the rule for sub(a,b) relies on the generalized rule for combinex(b). The knowledge that any pair of non-zero x terms in equation b should be combined had to precede the final refinement of the rule for sub(a,b). This demonstrates that the rules are independently generalized with the constraint that the order of training instances can influence the rate of learning.

Symbolic Integration

This section presents a short example of perturbation in the domain of symbolic integration. Following chapter 3, assume that there are only two operators in the domain:

$$OP1: \int x^n \, dx \rightarrow \frac{x^{n+1}}{n+1} + C$$

$$OP2: \int a \, poly(x) \, dx \rightarrow a \int poly(x) \, dx$$

OP1 integrates a term consisting of the variable of integration, x. OP2 extracts a constant, a, from the expression being integrated.

The first rule formed by episodic learning in chapter 3 is:

$$1 \; -- \; 7 \int x^2 \, dx \rightarrow OP1$$

This rule is overly specific and is generalized by PET.

First, perturbation operators are applied to generate a set of examples. Four of these examples are:

$$\underset{E_1}{\square \int x^2\,dx} \qquad \underset{E_2}{7 \int \square\, dx} \qquad \underset{E_3}{\boxed{8} \int x^2\,dx} \qquad \underset{E_4}{7 \int x^{\boxed{3}}\,dx}$$

E_1, E_3 and E_4 are classified as positive examples of the concept "states in which $OP1$ is effective." Since the current heuristic rule for $OP1$ has a score of 1, effectiveness is determined by a state transition to a goal state (a state not containing an integral). A minimal generalization of the state description of the current rule for $OP1$ with E_1, E_3 and E_4 yields the new rule:

$$1 - - \int x^{positive}\,dx \rightarrow OP1$$

where *positive* represents any positive integer.

Only one more teacher supplied training instance is required for PET to converge on the final heuristic rule for $OP1$. Given the positive example:

$$\int x^{-3}\,dx$$

PET mimimally generalizes to:

$$1 - - \int x^{nonzero}\,dx \rightarrow OP1$$

PET's Learning Rate

This section describes the learning rate of the PET system with perturbation. For this purpose, the learning rate is defined to be the number of training instances required to achieve concept convergence.

Concept convergence rate is a function of the complexity of the rule and the depth of the concept hierarchy tree. As demonstrated with the examples above, the initial rule learned by PET is overly-specific. The initial rule is based on the first

positive instance of the rule presented by the teacher. Assume that this instance has n descriptors. Further assume that the concept hierarchy tree used for generalizing descriptors is of depth d. (If there are multiple concept hierarchy trees then use the depth of the deepest tree for the worst-case analysis).

Assuming that no two training instances are identical, each positive training instance forces a generalization of (at least) one descriptor of the concept description. In the worst case, each generalization moves up a single link in the concept hierarchy tree. In the worst case, each descriptor must be generalized to the root node of the tree. Therefore, the number of training instances required is $n * d$.

Perturbation improves this learning rate by testing for irrelevant descriptors before generalizing relevant ones. If m descriptors are determined to be irrelevant, then the number of training instances required for concept convergence in the worst case is reduced to $m + d(n - m)$.

The learning rate is only partially influenced by the order of training instances presented by the teacher. This influence is significantly mitigated by perturbation which capitalizes on training. Irrelevant descriptors are removed and relevant descriptors are generalized even when the order of training instances is non-optimal. Unlike Winston's ARCH system [WINS75], the order of training instances presented to PET cannot result in an erroneous generalization. The worst case learning performance (analyzed above) is the most serious consequence.

Conclusions

This chapter discusses an approach to generalization which reduces the reliance of the learning element on the teacher. The technique, perturbation, automatically generates and classifies a set of examples of a concept. This set of examples is then provided to a "standard" induction algorithm to form a concept description.

Perturbation is motivated by the observation that the world is inherently regular. That is, if an example is classified as positive for a particular concept then other examples which are highly similar to it are likely to be positive as well. Perturbation generates this set of highly similar near-examples and near-misses by making minimal changes to an initial example.

Perturbation classifies examples by using the knowledge acquired by episodic learning. If a generated example plays the same role in an episode as the initial example did, then the generated example is classified as positive. One of the strong advantages of perturbation is that generation and classification of examples is quite efficient.

The chapter concludes with examples of perturbation applied to two domains. The effectiveness of perturbation at enabling concept convergence with minimal teacher assistance is demonstrated.

CHAPTER 5
Learning Operator Transformations

This chapter discusses a technique for learning what operators do. During problem solving, operators are applied to states to transit from one state to its successor. While learning problem solving, the learning element can see the states before and after the operator application. However, the transformation performed by the operator is hidden. By discovering the transformation the learning element can reason with the operator definitions and improve the learning process.

An algorithm is presented for learning operator transformations. The task is viewed as a form of learning from examples. The learning element is presented with examples of operators applied to specific states. From this, a representation called a *relational model* is formed which defines the general transformation performed by the operator. Examples of the PET implementation of the algorithm are given for the domain of symbolic integration. Chapter 6 continues this discussion by demonstrating the significant contribution of relational models to learning problem solving.

Defining the Problem

The world abounds with opaque operators. *Opaque* operator representations hide the sematics of operators. The transformation performed by the operator is concealed by the operator representation. Examples of opaque operators are familiar to everyone who has learned a task by observing experts who are proficient at the task. Their actions are recognized as legal but the observed solution seems magical. Their instruction cannot be fully assimilated without an understanding of the transformation performed at each step in the solution path.

One of the requirements for gaining expert problem solving skills is to acquire transparent representations for operators. *Transparent* operator representations reveal the "inner-workings" of the operator. This enables reasoning with operator definitions. Chapters 5 and 6 of this dissertation examine two issues:

1) how transparent operator representations can be learned from opaque representations (the subject of this chapter).

2) how transparent operator representations can improve the process of acquiring problem solving heuristics (the subject of chapter 6).

Transparent operator representations are essential for reasoning about operator transformations. For example, Waldinger's planning system [WALD77] demonstrates an effective use of transparent operator representations. The planner assumes that operators are represented with lists of pre-conditions, delete-conditions and add-conditions, *ala* STRIPS (see chapter 3). The planner solves multiple goals simultaneously and must address the problem of sub-goal conflicts. One approach to handling sub-goal conflicts is demonstrated by HACKER [SUSS73] and INTERPLAN [TATE75]. These planning systems simply backtrack when a conflict is encountered. A couple of goals are reordered and the planners try to solve the new problem description. However, Waldinger exploites the operator representation to discover goal conflicts **before** they arise in planning. Goal regression is used to back-up constraints from each of the goal description. If any constraints conflict, then the planner tries goal re-ordering. This ensures that the resulting plan is free of goal conflicts. As demonstrated by Waldinger, goal regression is a powerful reasoning strategy. But, as will be discussed in chapter 6, it relies on transparent operator representations.

This chapter discusses a technique for learning transparent operator representations from examples of operator applications. These transparent representations are called *relational models*. The central issue in learning relational models is the

utilization of existing "background knowledge" about the domain. Several techniques for incorporating background knowledge into an evolving concept description are discussed in the next two sections. This leads to the technique of learning relational models which is demonstrated with the PET system.

Related Work

This section discusses related work in learning operator representations. First, research in automatic programming is reviewed. A branch of this work, program synthesis from input/output pairs, is viewed as a form of learning from examples. Second, research in learning mental models is reviewed. This research seeks a cognitive model for learning operator semantics. Third, an induction algorithm by Vere is reviewed which experiments with learning in the presence of background knowledge. This is directly related to the PET approach of learning operator transformations in terms of existing domain knowledge.

Automatic Programming

One direction in automatic programming is program synthesis from input / output pairs. For example, given the pairs:

$$(\{a,b,c\},\{c,b,a\}),(\{e,f\},\{f,e\})$$

a program is constructed which inputs a list of atoms and outputs the list with the order of atoms reversed.

One approach adopted by researchers in automatic programming consists of the following two steps [BIER76]:

1) For each example input / output pair (ι, o), determine a transformation $\iota \rightarrow o$.

2) Find a program that performs this transformation when ι is input.

For the most part, step 1 has only been addressed theoretically. The space of possible transformations is large and discovering the transformation which is consistent with the training set is difficult. Researchers in program synthesis view the problem as intractable because of the following theorem:

> The programs for the partial recursive functions cannot be generated from samples of input / output behavior [GOLD67, BIER72].

Therefore, researchers have chosen to put the burden of specifying the transformation on the human programmer.

Step 2, on the other hand, is studied extensively and effective techniques for constructing programs exist [PRYW77, GREE76, FICK82]. This has led to a prevalent approach to program synthesis called *autoprogramming*. Autoprogramming skirts step 1 of automatic programming by requiring that the user specify the transformation to be performed using a high level language. Then autoprogramming synthesizes a detailed program which is semantically equivalent to the user's high-level description.

Automatic programming from input / output pairs is a form of learning from examples in which the concept description language is a formal programming language. Concepts can be expressed as horn clauses [SHAP83] or LISP code [SHAW75, HARD74]. For a complete automatic programming system, the problem of learning the transformation exemplified by the input / output pairs must be addressed. Relational models, the subject of this chapter, address the problem.

Mental Models

Mental models research is concerned with examining how people understand the world. The tenet of this research is that people represent domain knowledge with qualitative models which enable prediction. The ability to draw inferences from a state description is called *envisionment*. An example of envisionment by a mental model is demonstrated with Hayes research. Hayes [HAYE79] analyzed how

people predict when a liquid will flow, stand still, or spread into a thin sheet on a surface. From this analysis he constructed a mental model which describes the inferences that a person can draw from each of these different liquid states.

Envisionment in a domain is possible using mental models because they represent the semantics of operators in the domain. Many researchers have proposed that mental models are acquired by analogical reasoning [DOUG83, BURS83, GENT83, WINS81, CARB83]. Of these, Douglas and Moran [DOUG83] focus on learning operator semantics in the domain of text editing by reasoning from existing knowledge about typewriting. The difficulty with reasoning by analogy is that misconceptions arise. The learner must discern which features are relevant in an analogical match between the two domains. Preventing misconceptions during analogical reasoning is a central issue in research on reasoning by analogy.

Discovering operator semantics by analogy is a powerful technique when there is sufficient existing knowledge to reason with. The technique of learning relational models addresses the issue of acquiring this initial knowledge. An integrated learning paradigm which first learns relational models to build a foundation of knowledge and then shifts to learning by analogical reasoning would be an important contribution to research in machine learning.

Vere's Induction Algorithm

Vere's experiments with learning in the presence of background knowledge [VERE77] are directly related to learning relational models. When learning in a domain D, ***background knowledge*** is a body of facts which is separate from D but relates to the interpretation of D. Essentially, background knowledge is the existing body of facts that a learner employs in the task of acquiring new knowledge.

Background knowledge is incorporated into an evolving concept description when the induction process leads to ambiguities. For example, Vere addresses the problem of inducing general descriptions of poker hands from examples. Consider

the two "full house" hands:

$$2\clubsuit,\ 2\heartsuit,\ 4\diamondsuit,\ 4\clubsuit,\ 4\spadesuit$$

$$10\heartsuit,\ 10\diamondsuit,\ J\spadesuit,\ J\clubsuit,\ J\heartsuit$$

Using the turning-constants-to-variables generalization operator [MICH83], the following concept description of full house hands is formed:

$$r_1 s_1,\ r_1 s_2,\ r_2 s_3,\ r_2 s_4,\ r_2 s_5$$

where r_i and s_j are variables which bind to a card's rank and suit, respectively. Vere calls this generalization **deterministic** because no information is required beyond that present in the concept description.

By contrast, learning the concept of "straight" from the examples:

$$3\heartsuit,\ 4\diamondsuit,\ 5\heartsuit,\ 6\diamondsuit,\ 7\clubsuit$$

$$10\heartsuit,\ J\spadesuit,\ Q\diamondsuit,\ K\diamondsuit,\ A\clubsuit$$

requires additional knowledge. The generalization of these examples is:

$$r_1 s_1,\ r_2 s_2,\ r_3 s_3,\ r_4 s_4,\ r_5 s_5$$

which lacks the constraint that the ranks in a straight must be serial. This ranking information is considered to be background knowledge to the domain of poker hands. Specifically, the instance language descriptions of the card hands are augmented with $next(x, y)$ relations, as in:

$$3\heartsuit,\ 4\diamondsuit,\ 5\heartsuit,\ 6\diamondsuit,\ 7\clubsuit,$$
$$next(3,4),\ next(4,5),\ next(5,6),\ next(6,7)$$

Now the concept description is:

$$r_1 s_1,\ r_2 s_2,\ r_3 s_3,\ r_4 s_4,\ r_5 s_5,$$
$$next(r_1, r_2),\ next(r_2, r_3),\ next(r_3, r_4),\ next(r_4, r_5)$$

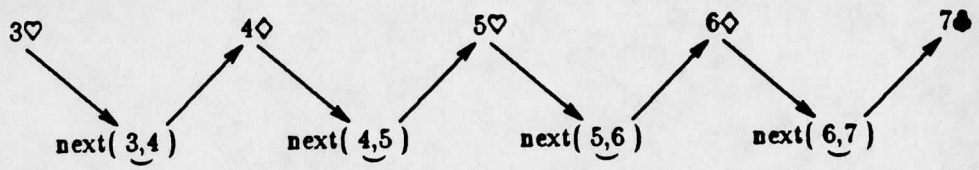


Figure 19

An Example Association Chain

Terms from background knowledge are "pulled-in" to an instance language description of an example to form an ***association chain***. Two literals, $L_1(t_{11}, t_{12})$ and $L_2(t_{21}, t_{22})$, in the description are ***associated*** if $t_{1i} = t_{2j}$. An association chain is a sequence of these associations. An example association chain for the example of "straight" given above is shown in figure 19. If a single chain links all literals which are associated (*i.e.* the association chain is unbroken) then the concept description is deterministic. A non-deterministic description is under-constrained, or over generalized.

The contribution of Vere's research is the identification of the problem of potential over-generalization and the use of background knowledge to solve it. The shortcomings of the approach are discussed in the next section which presents an early approach to learning with background knowledge in PET.

Rule Augmentation–The Precursor of Relational Models

This section discusses a technique for utilizing background knowledge for discovering general concept descriptions. The technique is called *rule augmentation* which refers to the addition of background knowledge to new knowledge represented as a production rule. Rule augmentation is demonstrated with examples from an early implementation of the PET system in the domain of simultaneous linear equations. Rule augmentation is then compared with Vere's use of association chains. Finally, a discussion of the shortcomings of rule augmentation motivates relational models for operator definitions.

Examples from Simultaneous Linear Equations

In the domain of simultaneous linear equations, PET uses rule augmentation to relate together terms in the instance language. For example, a relevant relation between coefficients is *product*(M,N,P) (the product of M and N is P). This relation augments the instance language. The augmentation represents necessary pieces of knowledge (not available at the surface level of the training instance) which a student must have in order to solve problems.

Augmentation of the instance language is necessary when the terms or values necessary for an operation (the RHS of a rule) are not present in the pre-conditions for the operation (the LHS of the rule). For example, the training instance:

$$a : 2x - 5y = -1$$
$$b : 3x + 4y = 10$$

might be presented with the teacher advice to multiply equation a by 3 with the operator mult(a,3). This yields:

$$a : 6x - 15y = -3$$
$$b : 3x + 4y = 10$$

From this training instance, PET forms the rule (after perturbation):

$$\{term(a, 2x), term(b, 3x)\} \rightarrow mult(a, 3).$$

Here the 3 in the RHS operation mult(a,3) appears on the LHS in term(b,3*x). In this case, the LHS of the rule is *predictive* of the operator on the RHS and no augmentation is needed.

In contrast, the teacher advice to apply mult(b,2) to the last pair of equations cannot generate a predictive rule. The operation is useful and yields:

$$a : 6x - 15y = -3$$
$$b : 6x + 8y = 20$$

The problem is that the 2 in the RHS operation mult(b,2) is not contained in the instance language description of the equations. Therefore, it could not be on the LHS of any rule in this language.

Relation	Semantics
sum(L,M,N)	(sum of L and M is N)
product(L,M,N)	(product of L and M is N)
square(M,N)	(square of M is N)

Table 3

Background Knowledge for Simultaneous Linear Equations

An augmentation of the instance language is needed to relate the 2 on the RHS with some term on the LHS. In this case, the additional knowledge needed is the 3-ary predicate *product*, specifically *product*(2,3,6). Now the rule to cover the training instance can be formed:

$$\{term(a, 6x), term(b, 3x), product(2, 3, 6)\} \rightarrow mult(a, 2)$$

This can be generalized (with more training instances) to:

$$\{term(a, N * x), term(b, M * x), product(L, N, M)\} \rightarrow mult(a, L)$$

where L, M and N are nonzero integers.

Concepts in the augmentation language form a second-order search space for generalizing to the correct rule for an operator. Concepts used by PET in the domain of simultaneous linear equations are listed in table 3. This is a (partial) list of concepts that a student might rely on for understanding relations between numbers. When a predictive rule cannot be found in the first-order search space then PET tries to form a rule using the augmentation as well. Concepts are pulled from the list and added to a developing rule. If the concept makes the rule predictive, then it is retained. Otherwise, it is removed and another concept is tried. If no predictive rule can be found then PET ignores the training instance.

Rule Augmentation in Perspective

The technique of rule augmentation has some important advantages over Vere's use of association chains. Vere describes an "association chain" which links

together each term in a rule. If a term in the rule is not linked in the chain then more background information must be "pulled in" until it is associated. This test for a break in the chain serves the same purpose as PET's test for rule predictiveness. Both tests detect gross over-generalization of a rule in which variables in the concept description are unconstrained.

One problem with both PET's and Vere's approaches to learning with background knowledge is determining how much knowledge to incorporate. Incorporating too little knowledge results in an over-generalized rule. However, detecting when too much knowledge has been pulled in is difficult. In this case, the rule formed will be over-specialized. PET overcomes this problem to a large extent by perturbation (see chapter 4). Vere relies solely on forming a disjunction of rules (each overly specialized) for the correct generalization.

Vere allows only one concept in the background knowledge. This further simplifies the task of knowing how much knowledge to pull in. However, as the complexity of problem domains increase, more background knowledge must be brought to bear. Rule augmentation addresses some of the problems of managing this knowledge.

But rule augmentation has two critical shortcomings. First, the predictiveness test used by rule augmentation is too weak. For example, consider the following (first) training instance for the multiply operator:

$$a : 3x - 4y = 9$$
$$b : 9x + 3y = 21$$

The teacher recommends the operator mult(a,3). Assuming that PET understands the advice (see chapter 3 on episodic learning), the rule after perturbation is:

$$\{term(a, 3x), term(b, 9x) \rightarrow mult(a, 3)$$

The problem is that the rule is "falsely" predictive. Although the rule passes the test for predictiveness, the 3 on the left hand side of the rule does not explain

(or account for) the 3 on the right hand side. Therefore, the rule will not correctly generalize. Since the predictiveness test is a simple syntactic check, it is easily fooled by coincidental surface level matches.

The second shortcoming of rule augmentation is that it does not represent the total transformation performed by the operator. Each term pulled in from background knowledge to augment a rule relates a literal on the left hand side of the rule with a literal on the right hand side. This process stops when the rule passes the predictiveness test. The idea of rule augmentation can be extended to define the transformation performed by operators. With this extension, background knowledge is used to relate **all** the terms in the description of the state before the operator is applied with terms in the description of the state after the operator is applied. Rather than simply relate a couple of terms, this approach defines a complete mapping. This extension of rule augmentation to learning operator transformations is discussed in the next section.

Relational Models

Relational models extend the idea of utilizing background knowledge to learn new concepts. In addition to augmenting concept descriptions to prevent overgeneralization, relational models represent the transformation performed by operators. This section formalizes the relational model representation of operators and heuristics used by PET. First, a variant of typical production rules for heuristics is introduced. Then, a formal definition of relational models is built on these heuristics. Finally, several examples of relational models learned by PET in the domain of symbolic integration are presented.

Representations for Heuristics and Operators

A relational model of an operator OP is built on a heuristic rule for OP. This rule is a variant of the production rule representation for heuristics presented in

chapter 3. In chapter 3, heuristic rules are of the form:

$$PRE \text{ state description} \rightarrow OP$$

with the interpretation:

If the current state, S, matches *PRE* then operator *OP* is recommended in S.

A relational model is of the form:

$$PRE \text{ state description } \overset{OP}{-} POST \text{ state description}$$

with the interpretation:

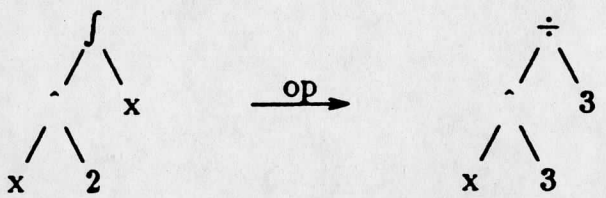
> IF the current state, S, matches *PRE*, **and** the state resulting from applying *OP* to *PRE* matches *POST* THEN *OP* is recommended in S.

The key difference between the two forms of rules is that the latter form explicitly represents the state description which results from the operator transition. This follows the style of rules proposed by Amarel [AMAR68]. One of the advantages of this style is that the representation helps to constrain inappropriate operator applications during problem solving. In addition to limiting operator applications to those states which match preconditions, Amarel-style rules require that the postconditions match as well. This form of heuristic rule thereby represents an entire transition.

In PET, these heuristic rules represent the *PRE* and *POST* state descriptions as parse trees. For example, the rule which recommends the operator

$$OP: \int x^n\,dx \rightarrow \frac{x^{n+1}}{n+1} + C$$

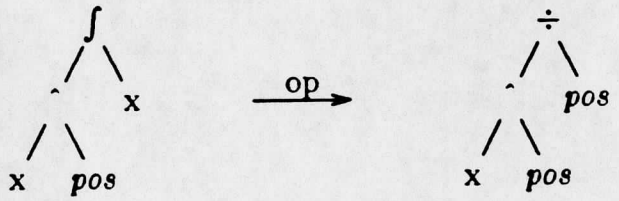
in state $\int x^2\,dx$ is:



where "$+C$" is dropped for simplicity. Note that the state resulting from the operator application, $POST$, is explicitly represented as the right hand side of the rule.

This form of heuristic rule is generalized using "standard" generalization techniques. For example, the generalization technique which is used with perturbation (chapter 4) forms generalizations of rules of the form $PRE \rightarrow OP$. Applying the same algorithm to states resulting from OP's application yields a generalization of $POST$. For each operator OP in a problem solving domain, PET uses the dropping conditions and climbing hierarchy tree generalization operators to induce general forms both for states in which OP is recommended and for states resulting from recommended applications.

However, this generalization scheme can yield unusable generalizations. For instance, the rule above can be generalized with the positive training example $\int x^3\,dx$. This yields the rule:



This rule is over-generalized since the critical relations are lost. There are two essential constraints in the original, instantiated rule that are lost in the generalization:

1) the x exponent in $POST$ is the increment of the x exponent in PRE.

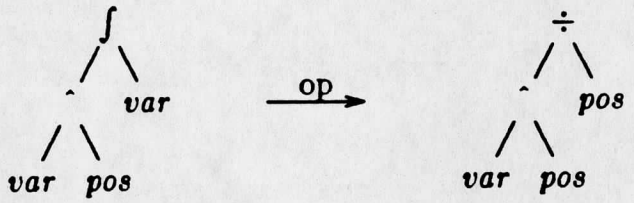
2) the denominator in $POST$ is the increment of the x exponent in PRE.

Further, generalizing with a third positive example, $\int y^4\,dy$, yields the generalization:

Relation	Semantics
equal(X,Y)	X and Y are equal
suc(M,N)	N is the integer successor of M
sumof(L,M,N)	sum of L and M is N
product(L,M,N)	product of L and M is N
power(L,M,N)	L raised to the M-power is N
derivative(M,N)	derivative of M is N

Table 4

Background Knowledge for Symbolic Integration

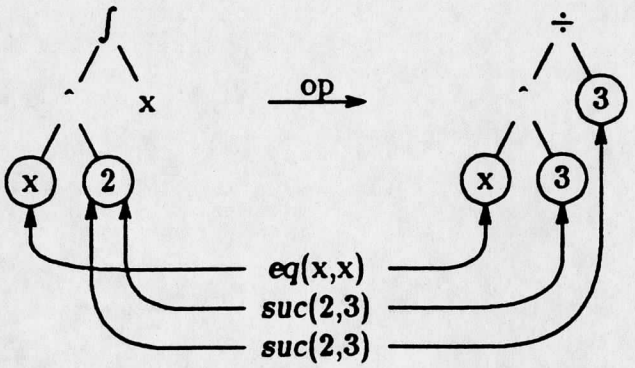


and a third essential constraint is lost:

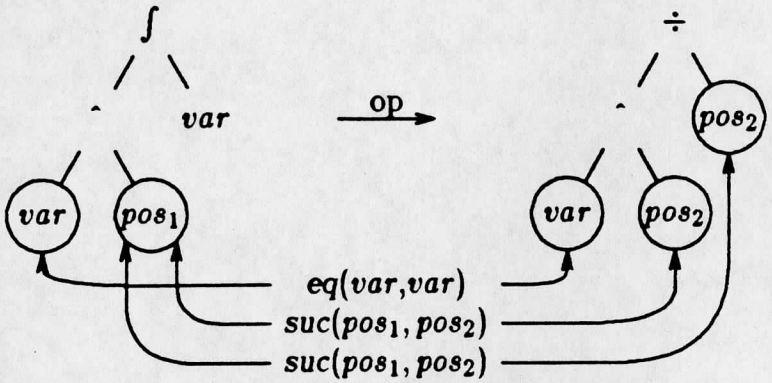
3) the variable in the numerator of *POST* is the variable of integration in *PRE*.

Relational models are an augmentation of this form of heuristic rule. An important role of this augmentation is to explicitly represent constraints between terms in *PRE* and *POST* so that they are not lost during generalization. Background domain knowledge augments the heuristic rules to relate terms in *PRE* with terms in *POST*. The previous section suggested some useful background relations for simultaneous linear equations. PET uses the set of relations listed in table 4 in the domain of symbolic integration.

The augmentation is a list of relations from background knowledge which is instantiated with terms from the heuristic rule. These relations represent constraints between the terms so that they are not lost during rule generalization. For example, an augmentation of the rule given above forms the following relational model:



Now, if the rule is generalized as above, the constraints are not lost. The generalized augmented rule is:



The augmentation alters the interpretation of an heuristic rule for operator *OP*. The augmentation is instantiated with terms from *PRE* and *POST* and the resulting rule is interpreted as:

> IF the current state, S, matches *PRE* and the state resulting from applying *OP* to *PRE* matches *POST* **such that** the relations in the augmentation hold, THEN OP is recommended in S.

With this intuitive understanding, relational models can be formally defined. A relational model is a 4-tuple $\langle OP, PRE, POST, AUG \rangle$. The augmentation, *AUG*, is a set of relations $\{rel_1, rel_2, \ldots, rel_n\}$ from background knowledge. Each relation $rel_i \in AUG$ has a relation name, or functor, and $m \geq 2$ arguments, $\{a_1, a_2, \ldots, a_m\}$. The purpose of the augmentation is to relate subexpressions of *PRE* with subexpressions of *POST*, thereby "linking" *PRE* to *POST*. To

establish these links, each a_j is constrained to be a subexpression of either *PRE* or *POST*, such that not all a_j are from the same source.

Actually, this is a simplification. By allowing a_j to be a subexpression of an argument of another relation in *AUG*, composites of relational descriptors can be formed by "daisy-chaining" a link between *PRE* and *POST* through multiple descriptors. For example, the relation that *PRE* is the double derivative of *POST* is represented by:

derivative(*PRE*,X),derivative(X,*POST*).

Learning Algorithm for Relational Models

This section presents an algorithm for learning relational models. The learning algorithm conforms to the model of learning from examples that permeates the design of PET. The input to the relational model learning element is an unaugmented heuristic rule. The learning element then searches for the "best" augmentation. The output of the learning element is a relational model in which the augmentation is instantiated with terms from the original unaugmented rule.

Augmentations are rated by an evaluation function ξ which estimates the "quality" of a relational model by measuring the coverage of *PRE* and *POST* by *AUG*. Intuitively, coverage is a measure of the number of nodes of *PRE* and *POST* which are in arguments of *AUG*. Formally,

$$\xi(\langle OP, PRE, POST, AUG\rangle) = \mid S_1 \mid + \mid S_2 \mid$$

where $\mid S \mid$ is the cardinality of set S and

$$S_1 = \{nodes\ n\ in\ PRE: \exists rel(a_1, a_2 \ldots, a_m) \in AUG \wedge \exists i, 1 \leq i \leq m, such\ that\ descendantof(n, a_i)\}$$

S_2 is similarly defined for nodes in *POST*. Note that an individual node in *PRE* or *POST* can contribute to coverage at most once since S_1 and S_2 are sets not bags.

Given an unaugmented heuristic rule $R = \langle OP, PRE, POST\rangle$, a relational model of R is constructed by searching for the set of instantiated augmentation

relations, *AUG*, which best covers R. This search is implemented in PET as a beam-search through the space of candidate augmentations. In this space, nodes are represented by the tuple $\langle AUG, Pool \rangle$ where *Pool* is the set of subexpressions of *PRE* and *POST* not covered by *AUG*. In particular, the initial state is $\langle nil, \{PRE \cup POST\} \rangle$. There is one operator in this search which is described by:

> Given a state $\langle AUG, Pool \rangle$,
> SELECT a relational descriptor, *D*, from the set of background concepts
> INSTANTIATE *D* with members of *Pool* or their sub-expressions
> REMOVE selected Pool members from *Pool*, yielding *Pool'*
> ADD instantiated descriptor to *AUG*, yielding *AUG'*.
> Generate new state $\langle AUG', Pool' \rangle$.

The search terminates with AUG when continued search fails to improve coverage.

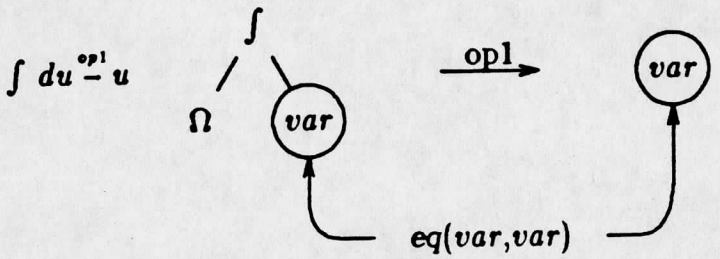
Built-in biases reduce the non-determinism of the search for an augmentation with maximal coverage and minimal complexity. In the selection of a relational descriptor, preference is given to more primitive relations, such as *equal* and *suc*, over more complex relations, such as *product*. Further, there are semantic constraints on the subexpressions selected to instantiate a relation. For example, the first parameter in the *derivative* relation must contain a variable of differentiation. Finally, note that the algorithm tries large subexpressions from *PRE* and *POST* before small subexpressions, thereby maximizing the coverage of the augmentation. If two relational models have the same coverage, then the one with fewer relations is preferred.

Examples of Relational Models

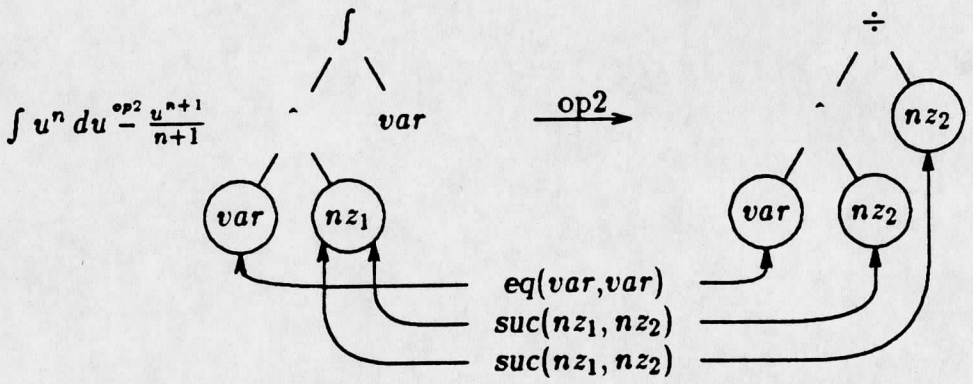
This section presents multiple examples of relational models in the domain of symbolic integration. These examples are generalized heuristics learned by PET and correspond to twelve operators used in symbolic integration [THOM68]. Each of the relational models uses generalized descriptors from the concept hierarchy tree

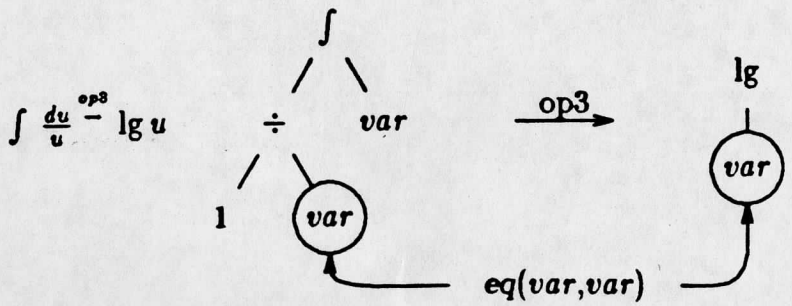
for symbolic integration (see figures 16 and 17 in chapter 4). Some of the examples are annotated with supplemental explanation.

The following relational model uses Ω to represent a null sub-tree and *var* to represent any variable (see the concept hierarchy tree).

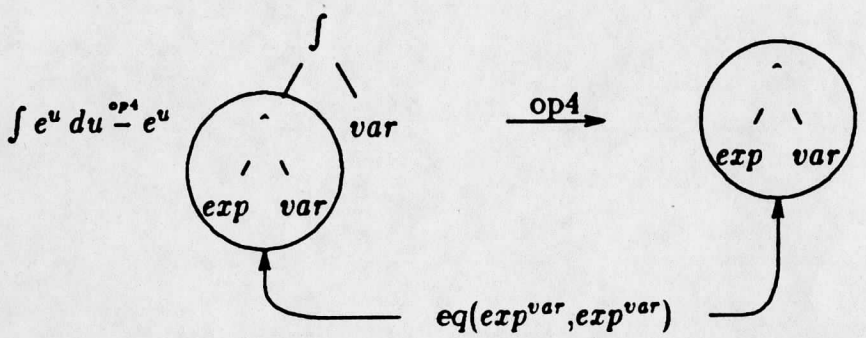


The next relational model uses *nz* to represent a non-zero integer from the concept hierarchy tree. Note that the descriptors use subscripts to enforce equality conditions.

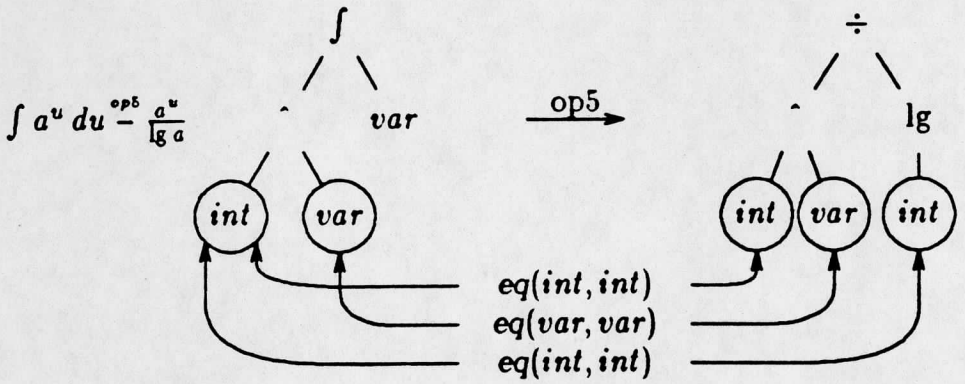


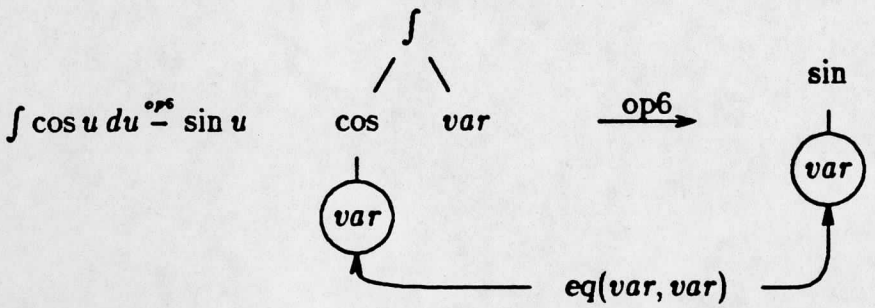


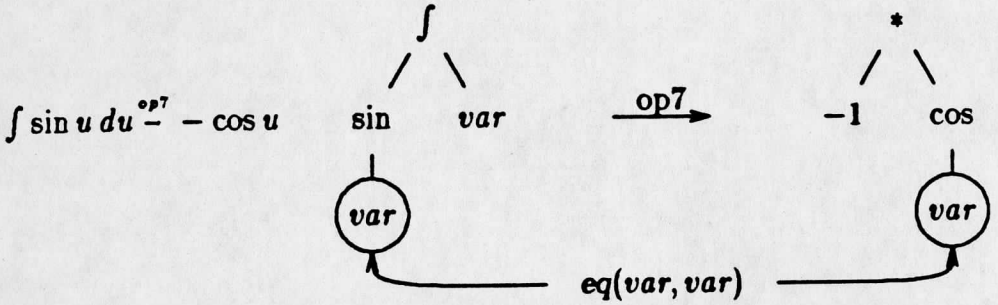
The next relational model illustrates PET's bias to select large sub-expressions of *PRE* and *POST* when instantiating augmentation relations. PET seeks an augmentation with maximum coverage of *PRE* and *POST*.



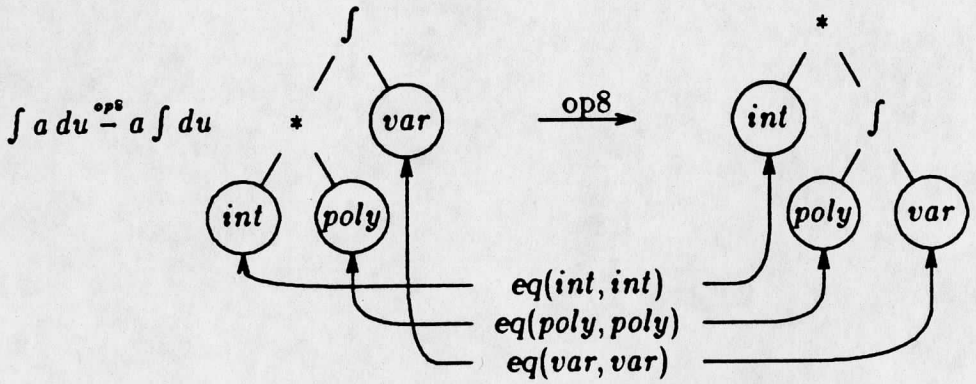
The generalized descriptor *int* in the following relational model represents any integer. Note that each occurrence of *int* must bind to equal valued integers for the heuristic represented by the relational model to recommend operator *op*5.

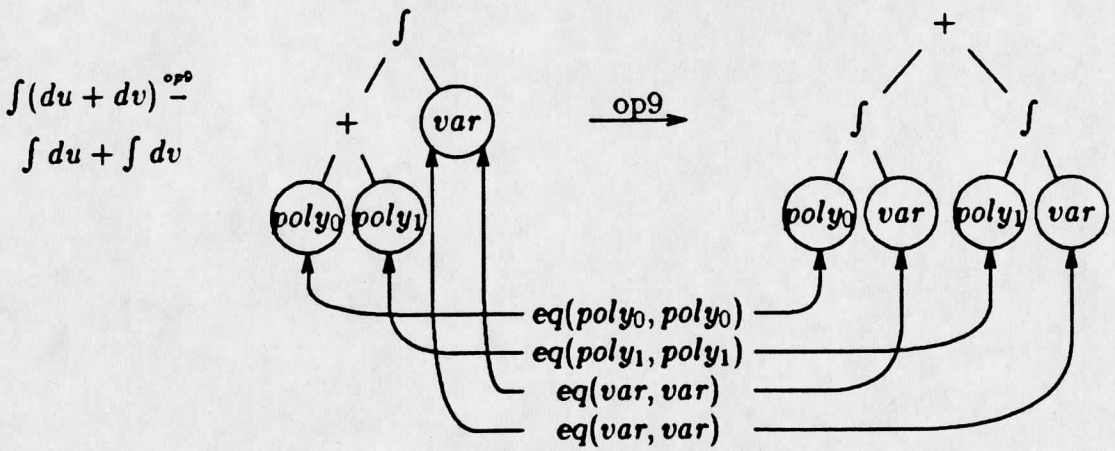


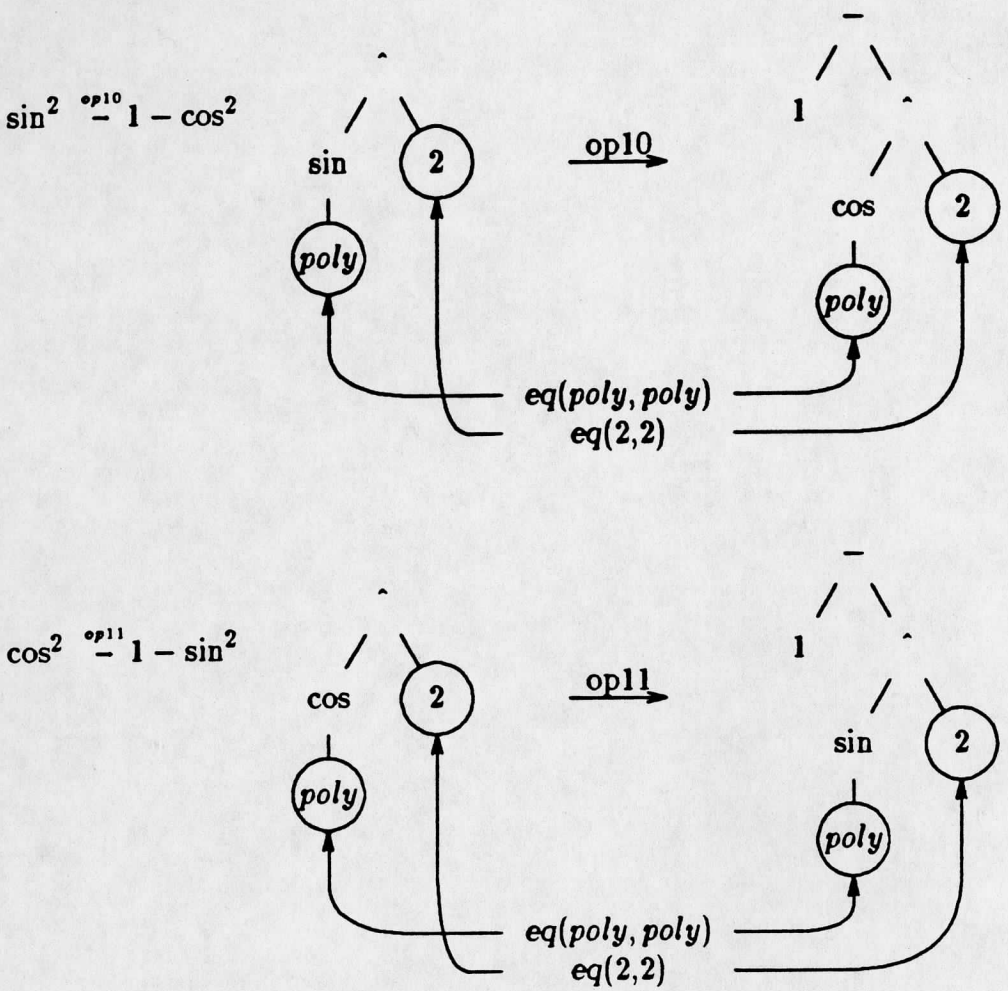




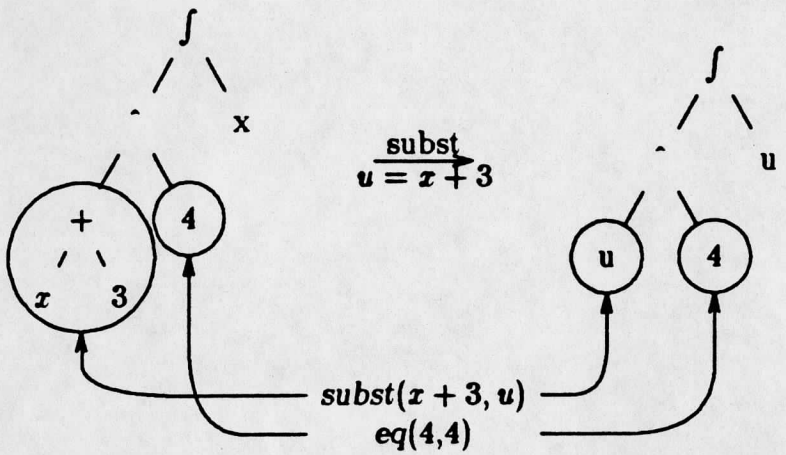
In the following relational model, *poly* represents any polynomial. Again, note the enforced equality constraints.





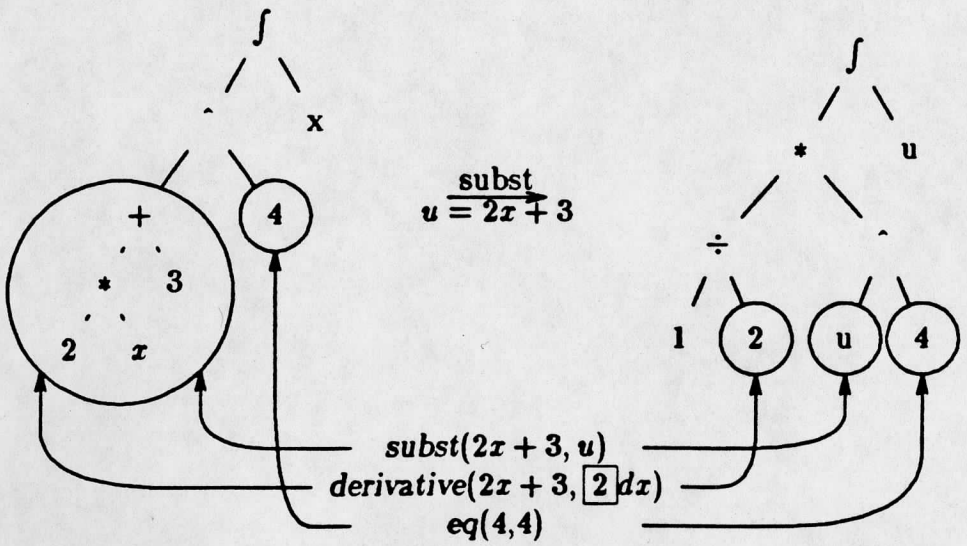


The following relational models are for the substitution operator. This operator simplifies a problem by replacing a subexpression with a variable. For example, the problem $\int (x+3)^4\,dx$ is simplified by replacing $x+3$ by the variable u. In this case, PET builds the relational model:

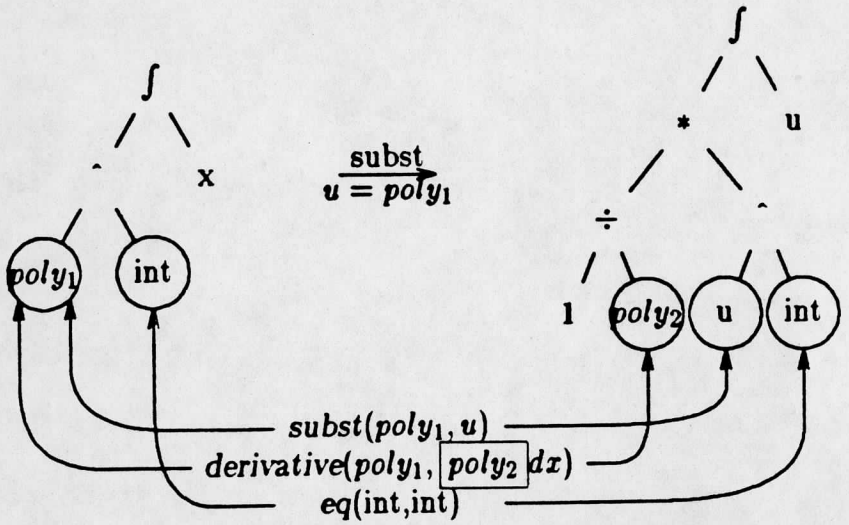


The augmentation descriptor *subst* is automatically added to the relational model by PET to record the substitution.

A more complex instance of the substitution operator is $\int (2x+3)^4\,dx$ with the advice to substitute u for $2x+3$. Unlike the previous example, the derivative of the substituted sub-expression is not 1. The resulting integral must be multiplied by the reciprocal of the derivative to maintain equality. The state resulting from the substitution is $\int \frac{1}{2}u^4\,du$. The relational model built by PET for the operation is:



PET generalizes this relational model using perturbation and further teacher training. The generalized rule is:



The relational model explicitly represents the constraint that $poly_2$ is the derivative of $poly_1$. This explicit representation overcomes representational inadequacies in LEX [UTGO83] which are discussed in the next chapter.

While not currently implemented in PET, it does not seem difficult for the learning element to note that the first relational model for the substitution operator shown above is a special case of the previous general rule. The augmentation of the general rule could guide the learning element. In particular, $derivative(x+3, \boxed{1}\, dx)$, $\frac{1}{1} = 1$ and $1 * poly = poly$. This simple reasoning permits some special case rules to be removed.

Conclusions

Augmentation of acquired knowledge is important to machine learning because it relates new knowledge to existing knowledge. Augmentation originates from a set of facts which relate to, but is separate from, the domain being learned. This body of knowledge is called background knowledge and is useful for interpreting new knowledge.

One important use of augmentation is learning relational models. Relational models represent the transformation performed by operators in a problem solving domain. The augmentation links features of the description of the state before an operator is applied with features of the description of the state after the application. This defines the transformation performed by the operator.

An algorithm is presented for learning relational models from examples of operator applications. The algorithm is a guided search for the augmentation which best represents the operator transformation. Multiple examples of relational models constructed by the PET learning system are presented.

CHAPTER 6
Improving the Learning Rate

This chapter demonstrates a powerful technique for rapid, independent learning. The technique is a variant of goal regression and is called constraint back-propagation. Constraint back-propagation enables the learner to detect generality in one rule of an episode which is used to guide the generalization of other rules in the episode. Using the technique, the learner efficiently refines the knowledge base with little teacher involvement.

Successful constraint back-propagation relies on an integration of the knowledge structures created by episodic learning, perturbation, and relational models. This integration demonstrates the soundness of the architecture of PET.

The chapter begins with a discussion of the general problem of improving the learning rate. Then constraint back-propagation in the LEX system is reviewed and operator representation is shown to be a central issue. The power of the relational model representation for operators is demonstrated with examples of rapid concept learning by PET.

The General Problem

This section discusses the general problem of improving the learning rate by constraint back-propagation. The learning rate of a learning system is roughly measured by the amount of knowledge acquired divided by the number of training instances used. The goal of the learning system is to maximize the learning rate. The method proposed in this chapter is to automatically generate training instances which are most useful in advancing the state of knowledge.

In terms of the PET system, the learning rate is improved by guiding the selection of perturbation operators. As discussed in chapter 4, perturbation partially automates the teacher's role in learning by examples. This is done by applying a set of perturbation operators to a single teacher supplied training instance. Each perturbation operator generates a slight variant of the original instance. While perturbation is effective for guiding the generalization process, it is unselective in the application of perturbation operators. The integrated approach to improving the learning rate proposed here addresses the problem of generating only the most useful perturbation candidates.

The technique of constraint back-propagation is similar to goal regression. Goal regression is a useful technique for reasoning backwards from a goal in planning. Utgoff applies the technique in the LEX system to learn new concepts and insert them in concept hierarchy trees. This related research is reviewed in the next section.

Related Work

This section reviews both the general technique of goal regression and a specific application of goal regression in machine learning. Goal regression is discussed in the context of an abstract planning system. Paul Utgoff's [UTGO83] research in adjusting concept hierarchy trees is reviewed as a useful application of a variant of goal regression, called constraint back-propagation. This review is useful for the next section which discusses the use of goal regression in PET.

Goal Regression

This section reviews the process of goal regression with examples from a blocks world planning system [RICH83]. Assume that there are two operators in the domain which are defined with the following STRIPS-like operator definitions:

STACK(x,y)
Preconditions: clear(y) ∧ holding(x)
Delete conditions: clear(y) ∧ holding(x)
Add conditions: armempty ∧ on(x,y)

PICKUP(x)
Preconditions: clear(x) ∧ ontable(x) ∧ armempty
Delete conditions: ontable(x) ∧ armempty
Add conditions: holding(x)

Goal regression is a technique for applying operator inverses. The regression of a goal state description through an operator determines what must be true before the operator is applied in order that the goal be satisfied afterward. For example,

$$\text{REGRESSION}(\overbrace{on(A,B)}^{goal}, \underbrace{pickup(C)}_{operator}) = on(A,B)$$

determines that the operator $pickup(C)$ will achieve the goal $on(A,B)$ when applied to any state in which $on(A,B)$ is true.

Goal regression is also useful for determining when a goal is unattainable by a particular operator. For example,

$$\text{REGRESSION}(armempty, pickup(A)) = \text{FALSE}$$

determines that there are no states in which the opertor $pickup(A)$ will achieve $armempty$.

Goal regression is a useful technique for reasoning with operator definitions. However, the representation of operators is critical to the success of the technique. The representation must make the pre and post conditions of the operator application explicit. This constraint is satisfied by STRIPS-like operator representations and relational models in PET. The next section reviews an application of goal regression to machine learning in which the issue of operator representations is of central importance.

Utgoff 's Application of Goal Regression

Utgoff's research [UTGO83] addresses the important issue of adjusting the bias that is inherent in learning from examples. The induction process is guided by generalization operators which determine legal generalizations of specific examples. The generalization operators represent biases, or prejudices, which enable concept descriptions to be found. If this bias is faulty then either incorrect concept descriptions are found or no concept description can be found at all.

Utgoff demonstrates his approach by adjusting bias which is built into the LEX system [MITC78, MITC83]. As reviewed in chapter 2, generalization in the LEX system is guided by concept hierarchy trees. Concept descriptions are found by applying the climb-hierarchy-tree generalization operator to specific examples. Utgoff proposes a technique for adjusting the concept hierarchy trees when the existing bias is faulty.

Utgoff's approach to adjusting bias consists of three steps:

1) Detecting when bias adjustment is needed – Adjustment is called for when a concept description cannot be found in the existing space of generalizations which is complete and consistent with the training set.

2) Determining what bias adjustment is called for – When the existing generalization language is determined to be inadequate, a new descriptor is created. A descriptor is sought which enables a concept description to be formed which correctly distinguishes between positive and negative examples of the concept.

3) Determining where the bias adjustment belongs – When a new descriptor is created, it must be correctly inserted in the existing concept hierarchy trees.

The second step in this approach to adjusting bias utilizes a variant of goal regression, which Utgoff calls *constraint back-propagation*. Utgoff utilizes constraint

back-propagation to discover new descriptors for the generalization language. New descriptors are created which correspond to the domain of an operator sequence. Each new descriptor is the composition of the preconditions of the heuristic rule for each operator in the sequence. This composition is formed by back-propagating the preconditions of each heuristic rule and collecting the regressed descriptors.

Constraint back-propagation relies on a representation for operators which makes explicit the transformation performed by the operator. In the case of STRIPS-like operator representations, the transformation is simply the set of state descriptors which are deleted or added by the operator application. These descriptors are explicitly listed in the delete condition list and add condition list of the operator definition.

However, the transformation performed by operators may not be so simply expressed. For example, there is an operator used in symbolic integration which replaces a sub-expression of an integral with a variable. In LEX, this operator is defined as:

$$\int poly(f(x))f'(x)\,dx \rightarrow \int poly(u)\,du, u = f(x)$$

where $poly(f(x))$ stands for a polynomial in x. The problem encountered by Utgoff is that back-propagation through this operator definition fails. The critical constraint that is not represented in this formalism is that whatever matches f' be the derivative of whatever matches f. This constraint is embedded in an opaque representation of the operator. This prevents constraint back-propagation since a semantically valid operator inverse cannot be found by the learning system.

The next section addresses the critical issue of transparent operator representations for successful constraint back-propagation. Constraint back-propagation is used to improve the learning rate of problem solving heuristics in the PET system.

Constraint Back-Propagation in PET

This section presents an approach to improving the rate of knowledge acquisition in the task of learning problem solving by example. The approach utilizes constraint back-propagation and is demonstrated in the PET system. First, the role of episodic learning, perturbation, and relational models in constraint back-propagation is discussed. Then, multiple examples of improved knowledge acquisition in the domain of symbolic integration are presented.

Integrating Episodic Learning, Perturbation, and Relational Models

This section describes a powerful technique for improving the learning rate of problem solving heuristics in the PET learning approach. This technique utilizes constraint back-propagation to automatically generalize heuristic rules in an episode. This technique is possible because of the successful integration of the components of the PET system: episodic learning, perturbation, and relational models.

The learning rate of problem solving heuristics is improved by reducing the number of training instances examined before an heuristic is fully generalized. As described in chapter 4, PET applies perturbation operators to a single teacher-supplied training instance to generate and classify multiple near-examples and near-misses. Perturbation automates part of the teacher's role, but not the task of *selectively* generating training instances which are most useful in enabling concept convergence. Constraint back-propagation presents an alternative to naïvely generating all possible training instances.

Using constraint back-propagation, perturbation candidates are generated which test features of a concept desciption which are believed to be overly specific. The selection of perturbation candidates is heuristically guided. This heuristic relies on knowledge which is represented by episodes and relational models. Specifically,

1) relational models represent the transformation performed by an individual rule application.

2) episodes represent the "chaining" of individual rules into a useful problem solving sequence.

Consider an episode E consisting of rule applications $r_1, r_2, \ldots, r_n$. Each rule r_i is represented with relational model $\langle OP_i, PRE_i, POST_i, AUG_i \rangle$. AUG_i represents the "intra-rule" links between PRE_i and $POST_i$. "Inter-rule" links are implicit in E. As described in chapter 3, r_i is added to an episode if it enables r_{i+1}. This establishes an implicit link between $POST_i$ and PRE_{i+1}. Constraints imposed on r_i by r_j, $i < j$, are discovered by following inter-rule links through E and intra-rule links through rules. These constraints suggest perturbation operators for r_i.

The heuristic of locating overly specific features by propagating constraints through episodes is motivated by this observation:

> Due to the incremental growth of episodes, for any pair of rules r_i and r_j, $i < j$ in E, the size of the training set for r_j exceeds the size of the training set for r_i because every training instance for r_i is also a training instance for r_j.

This suggests that features of PRE_j and $POST_j$ are more general than features of PRE_i and $POST_i$. PET selects perturbation operators which capitalize on this observation by back-propagating general features of PRE_j to potentially overly-specific features of PRE_i.

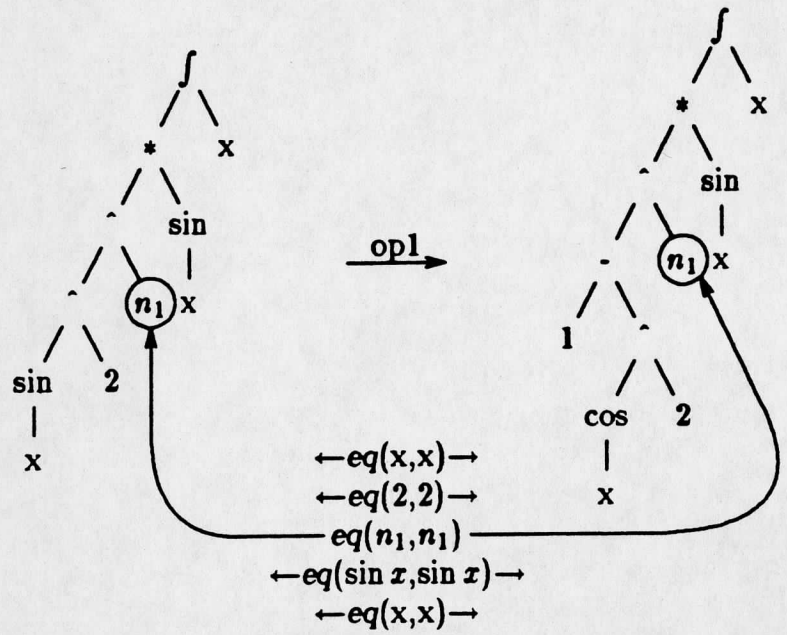
Examples from Symbolic Integration

Constraint back-propagation in the domain of symbolic integration is illustrated with an example from Utgoff [UTGO83]. The example demonstrates how PET automatically selects the single perturbation candidate from the enormous space of possibilities which enables useful concept generalization.

Assume that from prior training for operator

$$OP1 : \sin^2 x \rightarrow 1 - \cos^2 x$$

PET has acquired the following relational model:

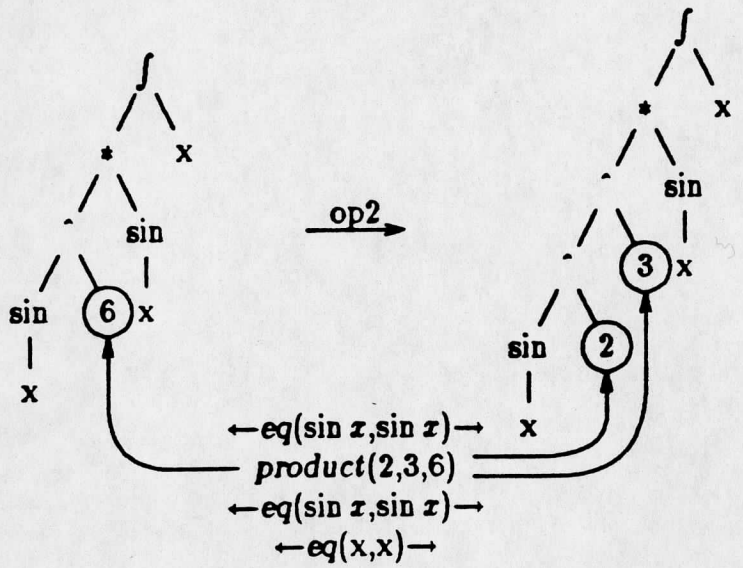


Note that this model has been generalized from ground instances such that PRE_{op1} matches states of the form $\int (\sin^2 x)^{nonzerointeger} \sin x \, dx$.

Now PET is presented the training instance $\int \sin^6 x \sin x \, dx$ with the advice to apply the opaque operator:

$$OP2 : \sin^n x \rightarrow (\sin^2 x)^{\frac{n}{2}}$$

PET applies the operator, yielding $\int (\sin^2 x)^3 \sin x \, dx$. As described in chapter 3, PET can only learn a rule for this training instance if it achieves a known (sub)goal (allowing the rule to be integrated into an existing episode). In this example, the training instance achieves the subgoal defined by PRE_{op1}. The following relational model for the training instance is built by the state-space algorithm in the previous section:



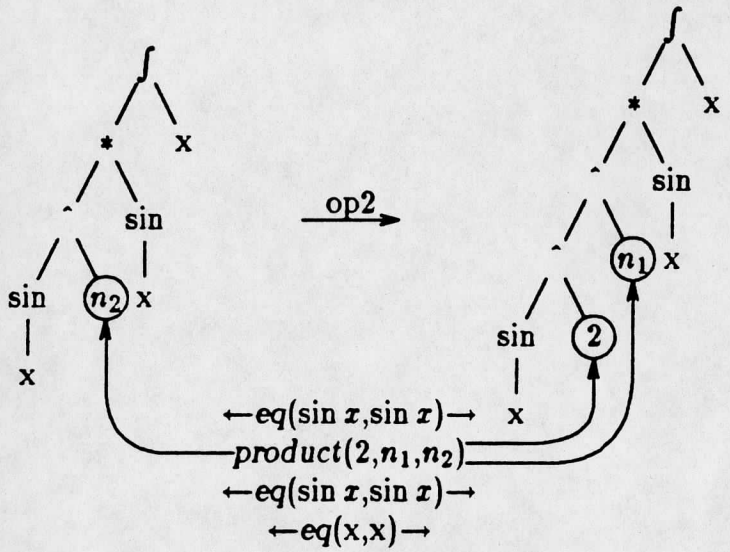
Now that episodic learning has associated the relational models for $OP1$ and $OP2$, perturbation operators are applied to generalize the model for $OP2$. The relaxed constraint in PRE_{op1} is regressed through the episode with the potential of identifying a feature of PRE_{op2} which can be relaxed (generalized). The inter-rule link implicit in episodes connects the relational model of OP2 with the relational model of OP1. Matching $POST_{op2}$ with PRE_{op1} binds variable n_1 with 3. This suggests that the relational model for OP2 is overly-specific. Perturbation tests relaxing this constraint by generating a training instance with the feature slightly modified. This is done by traversing intra-rule links represented by the augmentation. Specifically, PET generates a useful training instance by the following steps:

1. Locate the relation $r \in AUG_{op2}$ with argument of 3 from $POST_{op2}$. In this case, $r = product(2,3,6)$.
2. Perturb r to generate a slight variant, r'. This is done in three steps: First, replace the argument with a neighboring sibling in a concept hierarchy tree. In this case, replace 3 with 4. Second, locate an argument p in r such that p is a sub-expression of PRE_{op2} and replace it by free variable x. In this case, $p = 6$. Third, evaluate the resulting partially instantiated descriptor to uniquely bind x to p'. In this example, $p' = 8$ and r' =product(2,4,8).

3. Generate PRE'_{op2}, a perturbation of PRE_{op2}, by replacing p by p'. In this example, $PRE'_{op2} = \int \sin^8 x \sin x \, dx$

4. Classify PRE'_{op2} as an example or near-miss of a state in which *op2* is useful. As discussed in chapter 3, PRE'_{op2} is an example if apply(OP2,PRE'_{op2}) achieves the same subgoal as apply(OP2,PRE_{op2}). In this example, PRE'_{op2} is an example which achieves the subgoal of PRE_{op1}.

Finally, PET generalizes the original training instance with examples generated by perturbation. The following relational model is the minimal generalization of this (2 member) training set:



Note that the *product*(2,n_1,n_2) augmentation descriptor corresponds to the concept of *even-integer*(n_2). This is the correct constraint on the heuristic since this episode is only effective on integral expressions of the form $\int \sin^n x \sin x \, dx$, where n is even. As demonstrated with this example, constraint back-propagation through episodes and relational models can discover generalized descriptors which suggest new concepts. While it is not the focus of this research, these concepts could augment the description language for subsequent learning.

Using Relational Models to Remove Spurious Descriptors

Relational models suggest perturbation candidates for removing spurious descriptors from an evolving rule. A spurious descriptor for a heuristic which recommends operator OP is one which is irrelevant to the effectiveness of OP. Perturbation, as described in chapter 4, removes spurious descriptors. However, the search for spurious descriptors is unguided.

Relational models allow PET to selectively guide the search for spurious descriptors. Rather than test the relevance of every descriptor, PET heuristically selects candidates. Given relational model $\langle OP, PRE, POST, AUG \rangle$ the heuristic states:

> Descriptors of PRE which are not transformed by OP may be irrelevant to the rule recommending OP.

Those descriptors of PRE which are not transformed are exactly those linked by the *eq* relation to descriptors of $POST$. This heuristic identifies candidate irrelevant descriptors which can be tested with perturbation.

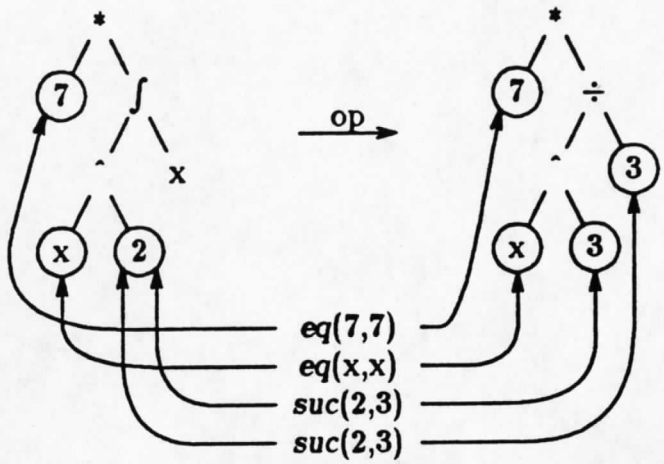
This process is illustrated with an example. Consider the training instance:

$$7 \int x^2 \, dx$$

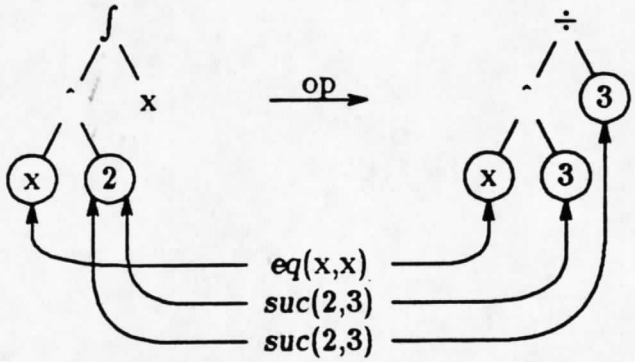
for the operator:

$$\int u^n du \stackrel{OP}{-} \frac{u^{n+1}}{n+1}$$

As described in chapter 5, PET forms the following relational model for this rule:



The heuristic for selecting perturbation candidates focusses on descriptors in *PRE* which are augmented with *eq* relations. With this guidance, perturbation generates two examples: $\int x^2\,dx$ and $7\int dx$. Of these, the first is classified by perturbation as a positive example and the second is classified as a negative example. The leading 7 is thereby recognized as a spurious descriptor and removed, yielding the generalized rule:



Of course, this technique is more useful on large rules with multiple spurious descriptors. Grossly overly specific training instances are not unusual in complex "real-world" learning domains.

Summary of Experience with Constraint Back-Propagation

This section summarizes the contribution of constraint back-propagation to improving the learning rate of PET. The main finding is that the number of perturbation candidates generated by PET can be drastically reduced without a drop in

knowledge acquisition. Without constraint back-propagation, perturbation generates every possible perturbation candidate. There are about a dozen perturbation candidates for an average training instance in symbolic integration.

Tests were conducted on reducing the number of candidates by using knowledge derived from constraint back-propagation. The tests found that the perturbation process can be constrained to generating at most five candidates. First, those candidates suggested by constraint back-propagation are tried. Then, those candidates which test for irrelevant descriptors are tried. Last, unguided perturbation is applied to randomly selected desciptors. The "resource constrained" perturbation process rarely resorts to unguided perturbation and achieves equivalent levels of knowledge acquisition as unconstrained perturbation.

Intelligent selection of training instances is of even greater consequence in "scaled-up" learning domains. Unguided perturbation becomes increasingly more unusable as the number of descriptors in rules and the branching factor of the concept hierarchy trees increases. When a learning element is knowledge-poor, unguided perturbation is a useful technique for acquiring initial knowledge. But, as more knowledge is acquired, the perturbation process can be focussed to improve the learning rate.

Conclusions

This chapter presents a technique for improving the learning rate of problem solving heuristics. The technique is based on goal regression and enables the learning element to back-propagate generalized constraints. Using the technique, the learner efficiently refines the knowledge base with little teacher involvement.

The technique arises from an integration of episodic learning, perturbation, and relational models. Perturbation automates part of the teacher's role in learning from examples, but not the task of *selectively* generating training instances which are most useful in enabling concept convergence. The learning rate is improved

by guiding the perturbation process to reduce the number of training instances generated. Heuristics for operators are represented with relational models and built into sequences with episodic learning. When a feature of a heuristic is generalized, the relaxed constraint is back-propagated through relational models and episodes to generalize other heuristics. By this technique, useful perturbation candidates are selected and the overall learning rate is significantly improved.

CHAPTER 7
Conclusions

This chapter summarizes the main contributions of this dissertation and concludes with suggestions for future work. The chief tenet of the dissertation is that learning to do problem solving requires acquiring multiple types of state space knowledge. Specifically, it requires knowledge of *when* operators are useful, *why* operators succeed, and *what* each operator does. The PET learning system demonstrates this capability in the domains of simultaneous linear equations and symbolic integration.

The PET learning cycle involves taking advice from the teacher and "flushing-out" the general lesson by experimentation. PET requests advice when knowledge acquired from previous problem solving does not apply to the current problem solving task. The teacher's role is to suggest useful state transitions to advance progress toward the goal. But, this advice is specific to the current problem and the PET learning element must discover the general lesson from the specific advice. This is done by forming hypotheses of the general lesson and testing them using the problem solver. When a hypothesis is confirmed it is integrated into the evolving knowledge base. Acquired general knowledge can then be used to direct problem solving on subsequent tasks.

A discussion of the limitations of the PET learning paradigm helps to put this research in perspective and to suggest future lines of research. Section 2 of this chapter describes five such limitations.

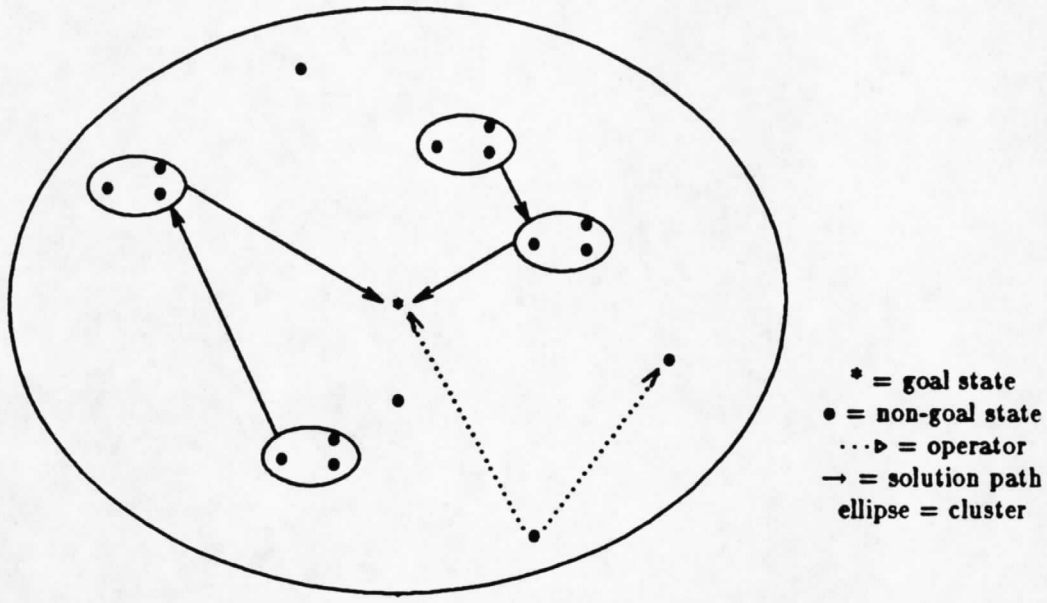


Figure 20

Knowledge of Solution Paths and State Clusters

The General Problem Re-Visited

This section reviews the contribution of the dissertation with respect to the general issues of learning problem solving discussed in chapter 1.

In a weak form, problem solving can be modelled as state space search. Viewed this way, a problem solving task is represented by an initial state (the problem to be solved), a set of goal states (solved problems), and a set of operators which can be applied to solve the problem. This defines a state space. Problem solving in this model consists of searching for a sequence of operators which transit from the initial state to one of the goal states.

The approach to learning problem solving proposed in this dissertation is to replace search by knowledge. The effect of this on problem solving is that operators are selected to apply to states based on knowledge of the search space rather that by trying alternatives with little domain knowledge to distinguish among them.

Essential domain knowledge consists of knowledge of solution paths, knowledge of heuristics from state clusters, and knowledge of operator transformations,

as shown in figure 20. This dissertation proposes a model for learning to do problem solving which focusses on these forms of knowledge. The integration of this knowledge yields a powerful learning paradigm which enables rapid learning from self-teaching. The model is demonstrated with a PROLOG implementation named PET.

Learning Solution Paths

The first form of essential domain knowledge is solution paths. A solution path is a sequence of operators which is applied serially and causes a transition to a goal state. This knowledge improves problem solving because search is eliminated. The solution path serves as a procedure which dictates the solution with no guesswork. Learning solution paths is the subject of chapter 3 which proposes a technique called **episodic learning**. Episodic learning builds solution paths by learning *why* each operator is useful - *i.e.* the role of each operator in solution paths.

Solution paths, or episodes, can be treated as units during problem solving. This permits the problem solver to concentrate on bigger pieces of the problem. Details within an episode can be ignored, thus freeing the problem solver from laboring over multiple local decisions. Knowledge of episodes is essential for efficient problem solving.

PET learns episodes by incrementally acquiring state evaluations. A state evaluation measures the relative quality of a state in the state space. This measure is determined by the distance (number of state transitions) from the state to a goal. This measure does not rely on a static evaluation function which merely *estimates* the quality of a state. Static evaluation functions suffer from the limitations of hill-climbing. That is, evaluations may be incorrect due to local maxima, ridges, or plateaus in the search space. Moreover, static evaluation functions for a domain are difficult to discover without extensive knowledge of the domain. This *a priori* knowledge cannot be assumed when the goal of the research is automated learning.

An episode is constructed from an operator sequence when the sequence causes a transition from one state to another state of higher quality. It is important to note that episodes "pass-over" local irregularities in the search space. That is, during the application of a sequence of operators, the sequence of states in the solution path may not appear to be monotonically approaching the goal. This, again, is a source of frustration for hill-climbing search. Episodes thus bridge expanses in the state space and reduce the complexity of problem solving.

Episodic learning proceeds by incrementally back-propagating state evaluations. This process begins with the goal states in the state space because their evaluation is known. Then, during problem solving, PET is advised to transit from other states to goal states. Consider a transition from state S_1 to state S_g, a goal state, via operator OP. PET assigns the state evaluation of one (distance to a goal) to S_1 Further, PET creates a heuristic rule $S_1 \rightarrow OP$ which records this transition. Since this solution path is recorded, S_1 becomes a learned subgoal in the state space. Now, knowledge of state evaluations, solution paths, and subgoals propagates back to states which reach S_1 in a single useful operator application.

Episodic learning constructs a lattice structure of solution paths. Each node of the lattice is a learned sub-goal. Arcs between nodes represent transitions which have proved useful in past problem solving. This knowledge of episodes is applied to new problem solving by navigating through the lattice starting with the initial state. Should the lattice not include the initial state, then weak problem solving methods are applied until a state is reached which is in the lattice. Problem solving with learned episodes replaces search with knowledge of past experience encoded in the lattice.

Learning State Clusters

The second form of state space knowledge required for efficient problem solving is state clusters. Clusters are groups of states which can be regarded as a single

state. This is critical to problem solving because the state space is potentially infinite. For problem solving purposes, clustering replaces an infinite set of states with a finite set of clusters.

Clusters are formed by grouping states which play the same role in a solution path. That is, if a solution path transits from $state_1$ to a goal then the group of states in the state space for which the same solution path is effective defines a cluster which includes $state_1$. Building state clusters is the subject of chapter 4 which discusses the technique of **perturbation**. Perturbation is an automated method of discovering *when* operators should be applied - *i.e.* the cluster of states in which each operator is effective at advancing toward the goal.

Perturbation is a general technique for guiding generalization. The role of perturbation in PET is to automatically generate a set of examples and near-misses of a concept. "Standard" generalization techniques are then applied to this set to find the general concept which is complete and consistent with respect to the training set.

Perturbation reduces a learning system's reliance on the teacher. By contrast, Winston's ARCH system [WINS75] is teacher-dependent (see discussion in chapter 2) The teacher must be cognizant of the current state of knowledge and the internals of the ARCH learning mechanism. The teacher then painstakingly generates training instances which advance the knowledge state. The training instances are classified by the teacher as either examples or near-misses of the concept being taught. The important constraint on training instances in ARCH is that they differ from the current concept description in only one or two features, avoiding the problem of finding the salient differences.

Perturbation focusses on the two roles of the teacher in learning by example: generating and classifying training instances. Perturbation automatically generates a set of training instances by making minor changes to a single teacher supplied instance. Each instance is automatically classified as an example or near-miss by

experimentation. That is, the learning element proposes a hypothesis concerning the instance which is then confirmed or denied by the performance element.

PET demonstrates the power of the technique of perturbation in "flushing-out" the general concept in a specific example of the concept. In a state space representation the example is a single state. The role of perturbation is to find neighboring states which are also examples of the general concept. This forms clusters of states in the state space thereby reducing the complexity of problem solving and learning in the space.

Learning Operator Transformations

The third form of state space knowledge which is critical to efficient problem solving is an understanding of operator transformations. There is a spectrum of operator representations from *opaque* to *transparent*. The world abounds with opaque operators which cannot be analyzed by the learning and performance system. A particular state to which an opaque operator is applied and the state resulting from the application can be observed. But the general transformation performed by the operator is hidden. Transparent operator representations, on the other hand, are unnatural in the world. These representations make explicit the transformation performed by the operator. The advantage of transparent representations is that a problem solver can reason with the operator "semantics." Since it is unreasonable to assume that operators are represented transparently, chapter 5 discusses a technique for learning transparent representations from examples of application of opaque operators. The representations learned are called **relational models**. Relational models explicitly represent *what* individual operators do.

PET constructs relational models by augmenting general operator descriptions with background knowledge. The background knowledge is in the form of domain specific relations. These relations are selected and instantiated with parts of each operator description. The non-determinism inherent in building relational

models is handled with a beam-search through the space of candidate augmentations. The search is constrained by built-in biases for the augmentation with the maximum coverage of the description with minimal complexity. Relational models serve to "tie-together" the operator descriptions into a cohesive statement of the transformation performed by the operator.

Integrating the Knowledge

The maximum capability of a learning system can only be realized if the sources of knowledge are united. Episodes, state clusters, and relational models provide the building blocks for an integrated learning system.

Chapter 7 discusses the contribution of each knowledge source to a common machine learning problem. The problem is to improve the learning rate by using existing knowledge to guide the refinement of new knowledge. This conforms to the observation (chapter 1) that knowledge is incrementally learned with the constraint that it's interpretation is biased by existing knowledge.

In the PET learning paradigm, this integration of knowledge improves the learning rate of problem solving heuristics. This is done by propagating knowledge of one state cluster to other state clusters through episodes. Episodic learning constructs useful solution paths. Perturbation generalizes each episode by forming clusters of states around each state in the solution path. Relational models enable PET to reason with the semantics of each operator in the solution path. Taken together, this knowledge is used by PET to guide the perturbation process to maximize the rate of self-teaching.

PET performs the propagation of knowledge through episodes by relying on links established by episodic learning and relational models. Episodic learning creates implicit inter-operator links between operators in a sequence. Relational models represent explicit intra-operator links between the "before and after" states of individual operators. This defines a complete path through each episode learned by

PET. PET uses these paths to propagate knowledge through the episode. At each cluster of states in the episode, PET applies perturbation operators to determine if the propagated knowledge can be used to refine the state cluster.

The integration of three sources of independently acquired knowledge enables PET to improve its learning rate without teacher involvement. This is done by using existing knowledge to guide the refinement of new knowledge.

Limitations and Extensions

This section discusses the known limitations of the PET paradigm for machine learning. Some of these limitations point to possible future extensions of this research. Most importantly, this serves to frame the set of issues addressed by PET and the domain of applicability of the paradigm.

Choice of Problem Domain

The choice of problem domain is problematic. On one hand, a problem domain should be selected which is well-defined and widely understood. This enables the researcher to focus on basic research issues and to communicate results to other researchers. That is, the domain should not conceal either the positive or negative research findings. Nor should the domain sidetrack the research from the central issues by imposing extraneous concerns. A good example of this type of problem domain is the blocks-world which has proved useful for research in natural language processing, planning, vision, and learning.

On the other hand, a problem domain should be selected which addresses hard, "real-world" problems. Recently, artificial intelligence has pushed away from "toy-domains" to face the challenges of natural domains. This trend has been quite useful to research in expert systems. But, in this case, a relatively well-understood knowledge representation (production rules) and knowledge applier (rule interpreter) exist to be built on.

The research in this dissertation is demonstrated with "simple" domains. The domains of simultaneous linear equations and symbolic integration are well understood and easily communicated. The domains aided, rather than distracted from, the development of episodic learning, perturbation, and relational modelling. However, mathematical domains are inherently suspect because of their similarity to programming [LENA83B].

With respect to problem domain, there are two directions that the research might evolve. First, because of the choice of problem domain, a critical comparison of PET and other machine learning approaches is possible. A comparison between PET and LEX might be particularly valuable since some of the same research issues are addressed differently in the same domain. Research in machine learning is quite spotty because there are few researchers and many problem domains. Addressing core research issues in similar domains allows for greater sharing of results.

A second direction is to apply the PET learning paradigm to a "natural" domain. Success at this task would demonstrate the general utility of the paradigm. One possible problem domain is flight simulation. The learning issue in this domain is to acquire knowledge of the operations required in a cockpit to achieve recognized goals. In this domain, episodic learning might apply to acquiring operator sequences. Perturbation might generalize these sequences by determining which features of state descriptions are critical to the success of the sequence. Relational models might represent the transformation performed by opaque operations in the cockpit. Finally, the simulator provides a responsive environment in which PET can learn by experimentation.

Weak Model of Memory

This dissertation does not propose a model of memory. A good memory model addresses the issue of integrating new knowledge with old knowledge so that

recall is efficient. Further, the model addresses generalization of experience which is essential for organization and recall.

One model which addresses a broad range of issues is the dynamic memory model by Schank [SCHA82]. Schank believes that learning is driven by expectation failures. New experiences are stored only if they fail to conform to one's expectations. Memory adapts to failed expectations by dynamically changing existing structures.

One important contribution of memory models is the hierarchy of knowledge required by the memory. This hierarchy permits efficient storage and recall of past experiences. The representation of knowledge used by PET is too flat to encode knowledge of multiple domains. Even the enormous quantity of problem solving knowledge required by chess (a single domain) would overwhelm an unstructured memory. One direction for future research is to develop memory models by examining the requirements for a complex learning task.

Using Relational Models to Guide Generalization

An essential part of machine learning is performing induction over a set of training instances. The result of the induction is a single generalization which is complete and consistent with respect to the training instances. Finding a generalization is hampered by the enormity of the space of candidate generalizations.

Relational models may help reduce the complexity of the search for a generalization. A relational model represents the transformation performed by an operator. In particular, the model shows the effect of the operator on the features of a state which are relevant to useful applications of the operator. This is exactly the set of features which must be generalized by the induction element.

Relational models decompose a group of features into small groups. The augmentation of the model represents the transformation of each group. Relational models constrain the generalization of two rules by requiring that the augmentation

of the rules match. This is a relatively simple step which defines a high-level, gross match of the two original rules. The details are filled in by matching corresponding features of the two rules. Relational models makes these correspondences explicit in the decomposition. This mitigates the problem of multiple matchings, thereby reducing the complexity of induction.

Viewed abstractly, relational models are useful during induction because they impose structure on the objects being generalized. This structure (augmentation) constrains the set of candidate generalizations by sharply reducing the possible matchings. Testing the feasibility and payoff of using relational models to guide generalization is a possible extension of this research.

Improving the Concept Description Language

It has long been recognized in machine learning that the concept description language greatly influences learning capability. Traditionally, researchers have painstakingly defined the description language so that the concepts that they wanted their system to learn were easily expressable. This shortcoming restricts the applicability of machine learning to those domains which are well enough understood to permit this heavy infusion of domain knowledge. In other words, machine learning is restricted to domains in which learning is not necessary.

As reported in chapter 6, there has been some research in dynamically redefining the concept description language [UTGO83]. The assumption of this research is that the description language should contain descriptors which are adequate for describing useful compositions of concepts. Or, in problem solving domains, the description language should enable descriptions of recurring operator sequences. The shortcoming of this prior work is that useful operator sequences are not explicitly learned. Also lacking is an explicit representation of operator transformations. The essential domain knowledge provided by episodic learning and relational modelling in the PET system may make a significant contribution to

learning concept descriptors. This research direction is briefly explored in chapter 6 of this dissertation, but more basic research is needed to address this critical issue in machine learning.

An Integrated Learning System

An explosive topic in machine learning is the design and development of an integrated learning system. This research attempts to combine multiple learning techniques into a single mechanism. These techniques might include learning by being told, learning by example, learning by experimentation, and learning by analogy.

One of the major investigations of such a study is the role that various learning techniques play For example, one might speculate that learning by example is most applicable when a learner is relatively knowledge poor. By contrast, learning by analogy might be extremely useful when a learner has significant knowledge to build on. Furthermore, a study of integrated learning addresses questions of knowledge structures: What should be the result from each learning technique? Is there a single knowledge representation? Can knowledge be represented at multiple levels?

PET might play an important role in an integrated learning system. PET demonstrates an effective combination of learning by being told, learning from example, and learning by experimentation. Further, PET constructs generally useful representations of domain knowledge. In an integrated system, PET's learning techniques would be "scaled-up" and contribute to a total learning paradigm with multiple levels of knowledge.

REFERENCES

[AMAR68] Amarel, S. On Representations of Problems of Reasoning About Actions. *Machine Intelligence 3*, D. Michie (ed.) (1968), 131–171, Edinburgh University Press.

[ANDE83] Anderson, J.R. Acquisition of Proof Skills in Geometry. Appearing in *Machine Learning*, Michalski, R.S., Carbonell, J.G., and Mitchell, T.M. (eds.), Tioga Publishing, 1983.

[ANGL78] Angluin, D. On the Complexity of Minimum Inference of Regular Sets. *Information and Control*, Volume 39 (1978), 337–350.

[ANGL83] Angluin, D. and Smith, C.H. Inductive Inference: Theory and Methods. *ACM Computing Surveys*, Volume 15, Number 3 (September, 1983), 237–269.

[ANZA78] Anzai, Y. Learning Strategies by Computer. *Proceedings of the Canadian Society for the Computational Study of Intelligence (CSCAI 2)* (1978), 181–190.

[BERL80] Berliner, H.J. Computer Backgammon. *Scientific American* (June, 1980)

[BIER72] Biermann, A.W On the Inference of Turing Machines from Sample Computations. *Journal of Artificial Intelligence 3*, 3 (1972), 181–198.

[BIER76] _____. Approaches to Automatic Programming. Appearing in *Advances in Computers*, Rubinoff, M. and Yovits, M. (eds.), Academic Press, Vol. *15*, 1976, pp. 1–60.

[BRAZ78] Brazdil, P. Experimental Learning Model. Appearing in *Proceedings of the Conference on Artificial Intelligence and Simulation of Behaviour (AISB)*, 1978, pp. 46–50.

[BUCH78] Buchanan, B.G. and Feigenbaum, E.A. Dendral and META-Dendral: Their Applications Dimension. *Artificial Intelligence 11* (1978), 5–24, North-Holland.

[BUND81] Bundy, A. and Silver, B. A Critical Survey of Rule Learning Programs. TR 169, University of Edinburgh, Dept. of Artificial Intelligence.

[BURS83] Burstein, M.H. Concept Formation by Incremental Analogical Reasoning and Debugging. Appearing in *Proceedings of the International Machine Learning Workshop*, June 22–24, 1983, Allerton House, Monticello, Illinois. Sponsored by The Office of Naval Research and the Department of Computer Science, University of Illinois at Urbana-Champaign, 1983, pp. 19–25.

[CARB83] Carbonell, Jaime G. Learning by Analogy: Formulating and Generalizing Plans from Past Experience. Appearing in *Machine Learning*, Michalski, R.S., Carbonell, J.G., and Mitchell, T.M. (eds.), Tioga Publishing, 1983.

[CLAN83] Clancey,W.J. The Epistemology of a Rule-Based Expert System. *Journal of Artificial Intelligence*, 20(3) (1983), 215–251.

[COHE82] Cohen, P.R. and Feigenbaum, E.A. *The Handbook of Artificial Intelligence*, Volume 1, William Kaufman Publ., 1982.

[DAVI77] Davis, R. Interactive Transfer of Expertise: Acquisition of New Inference Rules. *Proceedings of the International Joint Conference on Artificial Intelligence* (1977), 321–328.

[DELI79] Deliyanni, A. and Kowalski, R.A. Logic and Semantic Networks. *Communications of the Association for Computing Machinery*, 22 (1979), 184–192.

[DIET83] Dietterich, T.G. and Michalski, R.S. A Comparative Review of Selected Methods for Learning. Appearing in *Machine Learning*, Michalski, R.S., Carbonell, J.G., and Mitchell, T.M. (eds.), Tioga Publishing, 1983, pp. 41–81.

[DOUG83] Douglas, S.A. and Moran, T.P. Learning Operator Semantics by Analogy. Appearing in *Proceedings of the National Conference on Artificial Intelligence*, 1983.

[FICK82] Fickas, S.F. *Automating the Transformational Development of Software*, PhD Dissertation, Information and Computer Science Department, University of California at Irvine, 1982.

[FIKE71] Fikes, R. and Nilsson, N. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 2 (1971), 189–208.

[FIKE77] Fikes, R., Hart, P. and Nilsson, N. Learning and Executing Generalized Robot Plans. *Artificial Intelligence*, 3 (1972), 251–288.

[GENT83] Gentner, D. The Structure of Analogical Models in Science. Appearing in *Mental Models*, Gentner, D. and Stevens, S. (eds.), Hillsdale, N.J.: Erlbaum, 1983.

[GOLD67] Gold, E.M. Language Identification in the Limit. *Information and Control*, Volume 10 (1967), 447–474.

[GOLD78] _____. Complexity of Automaton Identification from Given Data. *Information and Control*, Volume 37 (1978), 302–320.

[GREE76] Green, C. The Design of the PSI Program Synthesis System. Appearing in *Proceedings of the Second International Conference on Software Engineering*, New York: IEEE, 1976, pp. 4–18.

[HAMP83] Hampson, S.E. *A Neural Model of Adaptive Behavior*, PhD Dissertation, Information and Computer Science Department, University of California at Irvine, 1983.

[HARD74] Hardy, S. Automatic Induction of LISP Functions. Appearing in *Proceedings of the Artificial Intelligence and Simulation of Behavior (AISB) Conference*, 1974, pp. 50-62.

[HAYE78] Hayes-Roth, F. and McDermott, J. An Interference Matching Technique for Inducing Abstractions. *Communications of the Association for Computing Machinery*, Vol. 21 (1978), 401–410.

[HAYE79] Hayes, P.J. The Naïve Physics Manifesto. Appearing in *Expert Systems in the Micro-Electronic Age*, Michie, D. (ed.), Edinburgh University Press, 1979.

[HEND79] Hendrix, G.G. Encoding Knowledge in Partitioned Networks. Appearing in *Associative Networks–The Representation and Use of Knowledge in Computers*, Findler, N.V. (ed.), Academic Press, 1979, pp. 51–92.

[HUNT66] Hunt, E.B., Marin, J. and Stone, P.T. *Experiments in Induction*, Academic Press, 1966.

[KUHN70] Kuhn, T.S. *The Structure of Scientific Revolutions*, Second Edition, Enlarged, The University of Chicago Press, 1970.

[LANG81] Langley, P. and Simon, H.A. The Central Role of Learning in Cognition. Appearing in *Cognitive Skills and Their Acquisition*, Anderson, J.R. (ed.), Lawrence Erlbaum, 1981.

[LANG83] Langley, P. Learning Effective Search Heuristics. Appearing in *Proceedings of the International Joint Conference on Artificial Intelligence*, 1983, pp. 419–421.

[LENA76] Lenat,D.B. *AM: An Artificial Intelligence Approach to Discovery in Mathematics as Heuristic Search*, PhD dissertation, Computer Science Department, Stanford University, 1976.

[LENA83] _____. The Role of Heuristics in Learning by Discovery: Three Case Studies. Appearing in *Machine Learning*, Michalski, R.S., Carbonell, J.G., and Mitchell, T.M. (eds.), Tioga Publishing, 1983.

[LENA83B] Lenat, D.B. and Brown, J.S. Why AM and Eurisko Appear to Work. Appearing in *Proceedings of the National Conference on Artificial Intelligence*, 1983, pp. 236–240.

[MCCA69] McCarthy, J. and Hayes, P.J. Some Philosophical Problems from the Standpoint of Artifical Intelligence. *Machine Intelligence 4* (1969), 463–502.

[MERV81] Mervis, C.B. and Rosch, E. Categorization of Natural Objects. *Annual Review of Psychology*, Volume 32 (1981), 89–115.

[MICH79] Michalski, R.S. and Dietterich, T.G. Learning and Generalization of Characteristic Descriptions: Evaluation Criteria and Comparative Review of Selected Methods. Appearing in *Proceedings of the Sixth International Joint Conference on Artificial Intelligence*, 1979, pp. 223–231.

[MICH80] Michalski, R.S. and Chilausky, R.L. Learning by being Told and Learning from Examples: an Experimental Comparison of the Two Methods of Knowledge Acquisition in the Context of Developing an Expert System for Soybean Disease Diagnosis. *Policy Analysis and Information Systems*, Vol. 4, No. 2 (June, 1980), 125–160. (Special issue on knowledge acquisition and induction).

[MICH83] Michalski, R.S. A Theory and Methodology of Inductive Learning. Appearing in *Machine Learning*, Michalski, R.S., Carbonell, J.G., and Mitchell, T.M. (eds.), Tioga Publishing, 1983.

[MINS63] Minsky, M.L. Steps Toward Artificial Intelligence. Appearing in *Computers and Thought*, Feigenbaum, E.A. and Feldman, J. (eds.), New York: McGraw Hill, 1963, pp. 406–450.

[MINS69] Minsky, M.L. and Papert, S. *Perceptrons: An Introduction to Computational Geometry*, Cambridge, Mass.: MIT Press, 1969.

[MITC78] Mitchell, T.M. *Version Spaces: An Approach to Concept Learning*, PhD Dissertation, Computer Science Department, Stanford University. (TR STAN-CS-78-711), 1978.

[MITC82] _____. Generalization as Search. *Artificial Intelligence*, Volume 18 (1982), 203–226.

[MITC83] Mitchell, T.M., Utgoff, P.E., Nudel, B. and Banerji, R. Learning by Experimentation: Acquiring and Refining Problem Solving Heuristics. Appearing in *Machine Learning*, Michalski, R.S., Carbonell, J.G., and Mitchell, T.M. (eds.), Tioga Publishing, 1983.

[NEWE61] Newell, A. and Simon, H.A. The Simulation of Human Thought. Appearing in *Current Trends in Psychological Theory*, Pittsburgh. University of Pittsburgh Press, 1961, pp. 152–179.

[NEWE72] _____. *Human Problem Solving*, Englewood Cliffs, NJ. Prentice-Hall, 1972.

[NEVE78] Neves, D.M. A Computer Program that Learns Algebraic Procedures by Examining Examples and Working Problems in a Textbook. *Proceedings of the Canadian Society for the Computational Study of Intelligence (CSCSI 2)* (1978), 191–195.

[NEVE81] Neves, D.M. and Anderson, J.R. Knowledge Compilation: Mechanisms for the Automatization of Cognitive Skills. Appearing in *Cognitive Skills and Their Acquisition*, Anderson, John (ed.), Lawrence Erlbaum Assoc., 1981, pp. 57–84.

[NILS80] Nilsson, N.J. *Principles of Artificial Intelligence*, Palo Alto: Tioga Publishing Co., 1980.

[OHLS82] Ohlsson, S. On the Automated Learning of Problem Solving Rules. Technical Report 10, Uppsala University, Computing Science Department, Uppsala, Sweden.

[PRYW77] Prywes, N.S. Automatic Generation of Computer Programs. Appearing in *Advances in Computers*, Rubinoff, M. and Yovits, M. (eds.), Academic Press, Vol. *16*, 1977, pp. 57–123.

[QUIN79A] Quinlan, J.R. Induction over Large Data Bases. Technical Report HPP-79-14, Heuristic Programming Project, Stanford University.

[QUIN79B] _____. Discovering Rules from Large Collections of Examples: A Case Study. Appearing in *Expert Systems in the Micro Electronic Age*, Michie, D. (Ed.), Edinburgh University Press, Edinburgh, 1979.

[QUIN83] _____. Learning Efficient Classification Procedures and their Application to Chess End Games. Appearing in *Machine Learning*, Michalski, R.S., Carbonell, J.G., and Mitchell, T.M. (eds.), Tioga Publishing, 1983.

[RICH83] Rich, E. *Artificial Intelligence*, New York: McGraw-Hill, 1983.

[ROSC76] Rosch, E., Mervis, C., Gray, W., Johnson, D., and Boyes-Braem, P. Basic Objects in Natural Categories. *Cognitive Psychology 8* (1976), 382–439.

[ROSE62] Rosenblatt, F. *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*, Washington, D.C.: Spartan Books, 1962.

[SAMM81] Sammut, C. Concept Learning by Experiment. *Proceedings of the Seventh International Joint Conference on Artificial Intelligence* (1981), 104–105.

[SAMU59] Samuel, A.L. Some Studies in Machine Learning Using the Game of Checkers. *IBM Journal of Research and Development 3* (1959), 210–229. (Reprinted in E.A. Feigenbaum and J. Feldman (Eds.) 1963, *Computers and Thought*. New York: McGraw-Hill, 71-105.)

[SAMU64] _____. The Banishment of Paper-Work. *New Scientist*, London (27th February, 1964). (Re-printed in AI Magazine, Vol. 4, No. 2, Summer 1983, pages 31-33.)

[SAMU67] _____. Some Studies in Machine Learning Using the Game of Checkers II–Recent Progress. *IBM Journal of Research and Development 11* (1967), 601-617.

[SCHA82] Schank, R.C. *Dynamic Memory: A Theory of Reminding and Learning in Computers and People*, Cambridge University Press, 1982.

[SCOT83] Scott, P.D. Learning: The Construction of a Posteriori Knowledge Structure. Appearing in *Proceedings of the National Conference on Artificial Intelligence*, August 22–26, 1983, pp. 359–363.

[SHAP83] Shapiro, E.Y. *Algorithmic Program Debugging*, (Based on PhD Dissertation, Computer Science Dept., Yale University, 1982), The MIT Press, 1983.

[SHAW75] Shaw, D.E., Swartout, W.R., and Green, C.C. Inferring LISP Programs from Examples. Appearing in *Proceedings of the Forth International Joint Conference on Artificial Intelligence*, 1975, pp. 260–267.

[SHOR76] Shortliffe, E.H. *MYCIN: Computer-based Medical Consultations*, New York: Elsevier, 1976. (Based on PhD Dissertation, Computer Science Department, Stanford University, 1974.)

[SILV83] Silver, B. Learning Equation Solving Methods from Worked Examples. Appearing in *Proceedings of the International Machine Learning Workshop*, June 22–24, 1983, Allerton House, Monticello, Illinois. Sponsored by The Office of Naval Research and the Department of Computer Science, University of Illinois at Urbana-Champaign, 1983, pp. 99–104.

[SIMO83] Simon, H.A. Why Study Machine Learning?. Appearing in *Machine Learning*, Michalski, R.S., Carbonell, J.G., and Mitchell, T.M. (eds.), Tioga Publishing, 1983, pp. 25–38.

[SUSS73] Sussman, G.J. *A Computer Model of Skill Acquisition*, PhD Dissertation, AI Laboratory, Massachusetts Institute of Technology (AI Tech. Rep. 297), 1973.

[TATE75] Tate, A. Interacting Goals and Their Use. Appearing in *Proceedings of the Forth International Joint Conference on Artificial Intelligence*, 1975, pp. 215–218.

[THOM68] Thomas, G.B. *Calculus and Analytic Geometry*, Fourth Edition, Reading, Mass.: Addison-Wesley, 1968.

[UTGO83] Utgoff, P. Adjusting Bias in Concept Learning. *Proceedings of the International Machine Learning Workshop*, June 22–24, 1983, Monticello, Illinois. Sponsored by The University of Illinois at Champaign-Urbana., 1983, pp. 105–109.

[VERE75] Vere, S.A. Induction of Concepts in the Predicate Calculus. *Proceedings of the Forth International Joint Conference on Artificial Intelligence* (1975), 281–287.

[VERE77] _____. Induction of Relational Productions in the Presence of Background Information. *Proceedings of the International Joint Conference on Artificial Intelligence*, 1977, pp. 349–355.

[WALD77] Waldinger, R. Achieving Several Goals Simultaneously. Appearing in *Machine Intelligence 8*, Elcock, E.W. and Michie, D. (eds.), New York: Halstead and Wiley, 1977, pp. 94–136.

[WATE70] Waterman, D.A. Generalization Learning Techniques for Automating the Learning of Heuristics. *Journal of Artificial Intelligence 1* (1970), 121–170.

[WHOR56] Whorf, B. *Language, Thought, and Reality*, Cambridge, Mass.: MIT Press, 1956.

[WINS75] Winston, P.H. Learning Structural Descriptions from Examples. Appearing in *The Psychology of Computer Vision*, Winston, P.H. (ed.), McGraw-Hill, 1975, pp. 157–209. (Based on PhD Dissertation, Computer Science Department, Massachusetts Institute of Technology, Cambridge, MA, 1970.)

[WINS77] _____. *Artificial Intelligence*, First Edition, Addison Wesley, 1977.

[WINS81] _____. Learning and Reasoning by Analogy. *Communications of the ACM 23* (1981), 689–703.

[WINS83] _____. *Artificial Intelligence*, Second Edition, Addison Wesley, 1983.

[YOUN77] Young, R.M., Plotkin, G.D. and Linz, R.F. Analysis of an Extended Concept-Learning Task. *Proceedings of the Fifth International Joint Conference on Artificial Intelligence* (1977), p. 285.