1991

Learning, Teaching, Optimization and Approximation

John R. Rice
Purdue University, jrr@cs.purdue.edu

Report Number:
91-032

LEARNING, TEACHING, OPTIMIZATION AND APPROXIMATION

John R. Rice

Computer Sciences Department
Purdue University
Technical Report CSD-TR-91-032
CAPO Report CER-91-11
June, 1991

To appear in
Expert Systems for Scientific Computing
(Houstis, Rice, Vichnevetsky, eds.)
North-Holland, 1992

LEARNING, TEACHING, OPTIMIZATION AND APPROXIMATION

John R. Rice
Department of Computer Science
Purdue University
West Lafayette IN 47907 U.S.A.
Technical Report CSD-TR-91-032
CAPO Technical Report CER-91-11

June 6, 1991

Abstract

A model of problem solving is given in terms of problem information, problem features, solution sets, solution selection forms, performance measures and performance requirements. Learning is then defined in terms of identifying problem features and selection forms. The selection forms include, for example, mathematical formulas, sets of rules, decision trees, exemplars, and neural networks. These forms have parameters that can be optimized and an additional aspect of learning is this optimization. Several examples are given and teaching is defined in this context. Lessons learned from the theories of optimization and approximation about learning to solve problems are summarized. The potential that supercomputers plus superoptimizers might create magic in learning how to solve problems is analyzed and the difficulties faced are discussed. There are closing comments on automatically computing geometrical features and rule sets.

Contents

1 Introduction

This paper is based on the viewpoint that *learning* is the acquisition of methods to solve problems. Further, the only methods considered are those that can be expressed computationally. This viewpoint may be limiting to some but it is broad enough to include both the classical and recent methodologies studied in computer science. We believe that almost all, if not all, learning is for the purpose of problem solving and that the informational aspects of all problem solving can be expressed computationally. This more philosophical question is not addressed here.

The topic thus is *learning methods to solve problems.* In Sections 3 and 4 we define problems and a model of the methods to solve them. In Sections 5 and 6 we define learning and teaching, and discuss briefly knowledge within this framework. We then reformulate learning in terms of approximation theory in Section 7 and discuss applicable lessons from optimization and approximation theory in Sections 8 and 9. The key question addressed in this paper is in Section 10: *Might supercomputers and superoptimizers allow us to learn how to solve (some, many, a few, most?) interesting problems?* Might neural nets, massively parallel machines, simulated annealing, genetic algorithms, etc., do this for us? The paper ends with observations about identifying geometric features and automatically generating rules.

Our answer to the key questions is *perhaps.* On the positive side we point to where new approaches and bigger computers in the 1960's revolutionized geometric modeling. These approaches are a simple instance of ideas being proposed for learning in general. On the negative side we present an example of a smooth, simple function of many variables where *none* of the new ideas and big computers help at all. The fundamental uncertainty is that we still have a very vague understanding of the nature of the problems we want to solve, we do not have measures of their difficulty which are useful in a practical way. We suspect that some interesting problems are easy enough that supercomputers and superoptimizers applied blindly will, almost by magic, learn how to solve them. We suspect that we can learn how to solve a much larger number of interesting problems by applying supercomputers and superoptimizers cleverly.

2 Structure of Problem Solving

We first describe the components of a problem and then give a model of the problem solving process.

2.1 Definitions of a Problem

We define a problem to have four components as follows, only one of which is the problem itself.

Properties: Numerical, symbolic, geometric or logical values that can be determined from the problem information.

Solution: One item from a specified set of possibilities.

Performance Function: An algorithm to compute values of performance parameters of a solution.

Requirements: Values specified for the performance parameters.

This definition is intended to include classes of closely related problems, e.g., *How long to cook a roast?* as well as *How long to cook the roast I have in my hand?* This definition does not require a unique assignment of information to a particular component. Thus the qualification "rare" in the problem *How long to cook this roast rare?* can be part of the problem properties or one of the requirements for a solution.

2.2 A Model of the Problem Solving Process

Figure 1 shows the diagram used to model the problem solving process and shows the four components of the problem. The problem properties and requirements are inputs to the process. The specified solution set and performance function are used to evaluate the solution produced and provide a grade. At this point the grade would be *accept* or *reject*, later we will see that a range of grades is useful in discussing learning to solve problems. Note also that, unlike in mathematics, most real problems have many solutions. We often speak of one solution being better than another.

Figure 1 identifies the two key ingredients of problem solving: features and selection forms. The *features* are the problem information items actually used to select the solution. These might be considered (and might be) problem properties but often they are not easily identified or computed. It is useful to keep the "raw" problem properties separated from the problem features because identifying the right features is often the most important step in learning to solve a problem. The *selection form* is the "thing" that is used to actually select the solution. This form is computational, a formula, a program, a decision tree, etc. An unusual name is chosen here to avoid prejudging one's view about the selection form's nature. It is a computable function (in the mathematical sense) whose arguments are the problem features and whose range is the solution set.

It is important to note that many problems have an enormous amount of problem properties: a photographic image may have millions of pixels to input to a scene analysis problem. A Fortran function may have thousands of characters (its code) or billions of numbers (its values at points) to input to a numerical quadrature problem. Rarely does one want to process all these properties in the selection form and thus feature identification is often essential.

Further, the selection form has parameters of several types which can be varied to affect the selection. A second important step in learning to solve problems is to vary these parameters. It is the optimization of these parameters that connects learning to solve problems with the theories of optimization and approximation.

3 Examples

Three examples are given to illustrate the ideas of the previous section and to show the flexibility of the definitions.

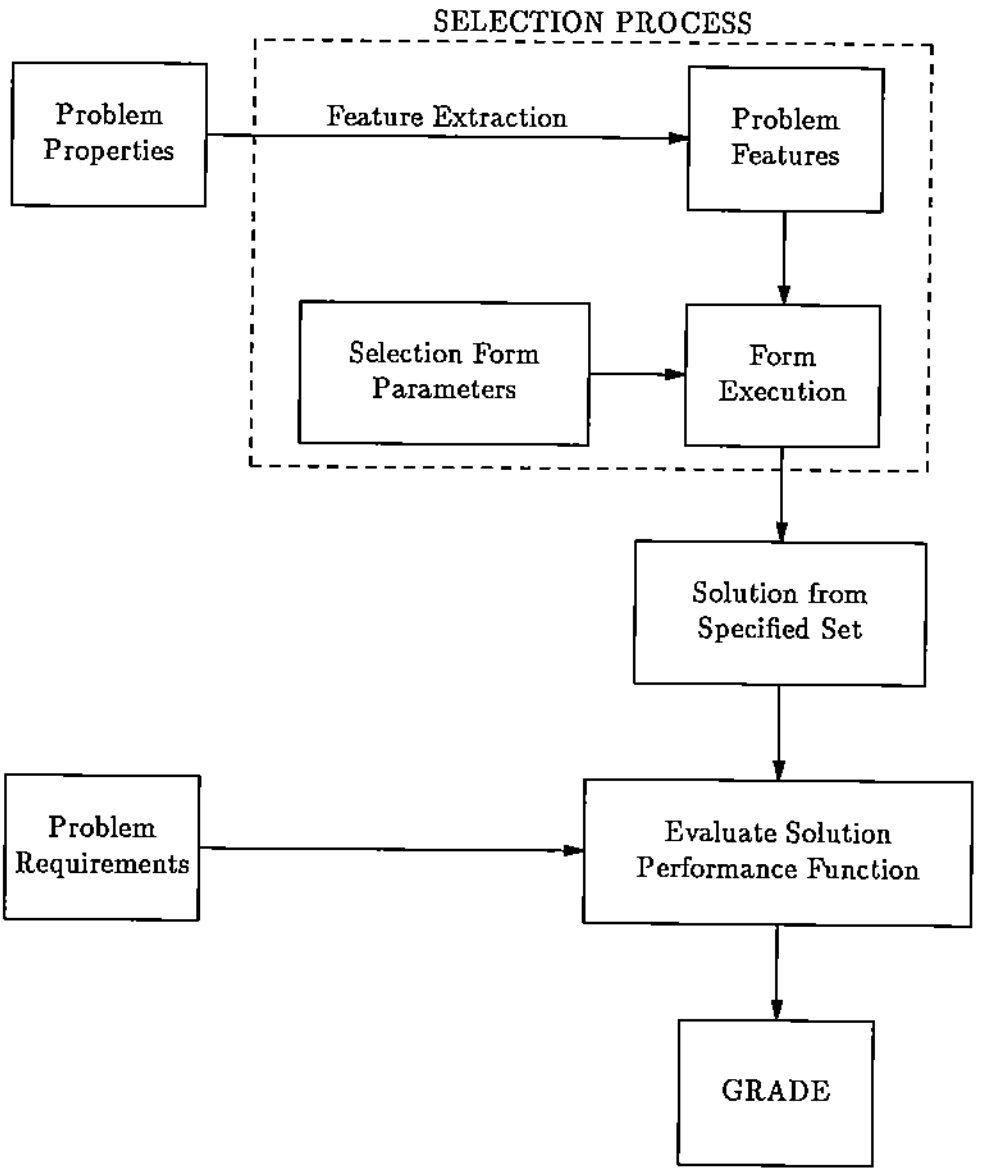

Figure 1: The model of the problem solving process. The inputs are problem properties and performance requirements which produce a solution with a grade (e.g., acceptable).

3.1 Cooking a Roast

A common problem in the kitchen is how long to cook a roast. A cookbook usually provides the solution in terms of two problem properties: the type of roast and its weight. For each roast type one finds instructions for the cooking time. A typical description is:

Weight (lbs)	2	3	4	5
Hours	2	3	3 1/2	3 3/4

Subtract 1/2 hour for rare, add 1/4 hour for well done. This is a verbal description of a piecewise linear function of cooking time versus weight (if one assumes linear interpolation is intended).

In this simple case we see that the features used are the two problem properties (type and weight) and that the selection form is a list of piecewise linear functions plus adjustments for rare and well done. The parameters are the (weight, time) values at the break points plus the adjustment times. These parameters have been learned from historical experimental observations. It is curious to note that there is only rough agreement between cookbooks in their solutions and cookbooks are commonly inconsistent in their specification for certain values. Perhaps this is why cooking is an art and not a science.

3.2 Controlling a Moving Object

This example is a classical mathematically defined problem. The laws of motion may be formulated as

$$\frac{d\mathbf{x}}{dt} = f(\mathbf{x}, t)$$

where $\mathbf{x}$ is an object's location in 3-space, t is time and $\mathbf{f}(\mathbf{x}, t)$ is the force applied to the object at time t. Thus $\mathbf{f}(\mathbf{x}, t)$ could represent the forces of wind, gravity and a jet engine on a rocket. If one wants to guide the rocket to a particular place at a particular time, one adds a *control force* $\mathbf{u}(\mathbf{x}, t)$ so that

$$\frac{d\mathbf{x}}{dt} = \mathbf{f}(\mathbf{x}, t) + \mathbf{u}(\mathbf{x}, t)$$

and the problem is: given $\mathbf{x} = \mathbf{x_0}$ at time t_0, given $\mathbf{f}(\mathbf{x}, t)$, then determine $\mathbf{u}(\mathbf{x}, t)$ so that $\mathbf{x} =$ desired position $\mathbf{x}^*$ at $t =$ desired time, t^*.

In this highly idealized example, the problem properties are also the features and are $\mathbf{x}_0$, x^*, t_0, t^* and $\mathbf{f}(\mathbf{x}, t)$. These are many treatises e.g., [ref 1] describing approaches to solving this problem, some of which are rather precise selection forms with mathematically defined parameters. A selection form is given in the next section in terms of neural networks for a particular instance of this problem.

3.3 Choosing a Method to Solve a Partial Differential Equation

Consider a partial differential equation (PDE) problem represented as

$$\begin{aligned} Lu &= f \quad \text{for} \quad u \in \mathbf{R} \\ Bu &= g \quad \text{for} \quad u \in \partial\mathbf{R} \end{aligned}$$

We assume the reader is familiar with PDEs, if not consult [1], [12] for terminology and background. The general problem is: *Given a PDE problem, select a method to solve it.* This is an example of the *algorithm selection problem* defined in [11]. Typical problem properties are:

linear	2 dimensional
Dirichlet boundary conditions	wave front present
elliptic	$f(\mathbf{x})$ expensive to evaluate
coefficient of u_{xx} = constant	singularity level 75%

In [12] an elliptic PDE problem interface is defined with 45 properties: 3 subroutines, 2 functions, 15 real numbers, 4 integers, and 21 logical variables. Furthermore, 36 features are defined which can, in principle, be computed even though they are quite expensive to estimate accurately. Of these, 18 relate to the PDE problem as a whole, 8 are specific to the operator L, and 10 are specific to the solution u.

The performance function has three components.

Error performance	=	$-\log\|\|\text{error}\|\|/\|\|u\|\|$,
Cost performance	=	Dollars charged for computer time,
Response performance	=	Hours waiting to obtain solution u.

As is common in more realistic problem solving, these performance measures are completely incommensurate and the performance function simply computes a vector of three values.

A method to solve a PDE also has three components as follows,

Numerical Method: A combination of numerical techniques, for example:

1) Ordinary finite differences Cuthill-McKee reordering Band Gauss elimination	2) Galerkin with cubic splines SOR iteration

Spatial Discretization: A particular way to discretize space and time, a mesh or grid or triangulation.

Implementation: A software system and a computer that implement the numerical method and spatial discretization.

The number of possible methods is incredibly large, even in realistic instances it can be enormous. In the ELLPACK system it is estimated that for many typical problems there are roughly 10,000 choices for numerical methods, 5,000 choices of discretizations, and 6 choices for implementation giving a solution set with 300,000,000 items. Restricting the granularity some and other similar simplifications can reduce the size of the solution set to perhaps 600,000 without a significant loss in performance. See [2] and [4] for further discussion of this matter. We estimate that in the experimental Parallel ELLPACK system [7] the number 600,000 increases to 10 million even though the number of available numerical methods is currently much smaller.

A selection form for this problem would be called an expert system unless it were very simplified. Three such systems have been studied. The first uses rules similar to Prolog (OPS 5 is actually used), see [2]. The second uses essentially the same knowledge, but its processing is carried out using Dempster-Shafer reasoning [4]. The third study uses an exemplar based expert system involving a very large data base of experimentally obtained performance data [8]. The exact nature of these selection forms is unimportant here, we only need to know that they use very complex computational methods.

4 Selection Forms

The examples of the previous section suggest a wide variety of selection forms is in use. A more systematic survey of common forms is given along with some examples.

4.1 Classes of Selection Forms

4.1.1 Mathematical. Mathematical formulas of various types are the classic solutions for problems and, indeed, one can argue that many of the special functions of mathematics are studied because they are useful in solving certain important problems. Examples of classical selection forms are:

$$\begin{aligned} \text{Answer} &= a_1 + a_2 x + a_3 x^2 + a_4 x^3 \\ \text{Answer} &= a_1 e^{-3x+t} + a_2/(1 + xt^2) \\ \text{Answer} &= a_1(\text{WEIGHT}) + a_2(\text{IQ}) + a_3\ (\text{SOCIO-ECON-CLASS}) \end{aligned}$$

Here x, t, WEIGHT, IQ, and SOCIO-ECON-CLASS are all problem properties or features and the a_i are selection form parameters. Less obvious but more important parameters here are the *basis functions*: e^{-3x+t}, $1/(1+xt^2)$, 1, x, x^2, x^3 and the number of them to be used.

More modern mathematical selection forms are the piecewise polynomials and splines discussed in Section 9. Random number generators are also used at times when one is unable to make a more rational choice.

4.1.2 Rule Sets. An intuitively plausible approach to selecting a problem solution is to apply some rules. This corresponds to practice in many common situations. This approach has been systematized in recent years as one of the original approaches to creating expert systems. One

might say that the *Prolog* language was invented for this purpose. For difficult problems (such as selecting a method to solve PDEs discussed earlier) the rules are often elected from "experts" in the problem area. These experts are thought to use a wide variety of rules in their problem solving which they do not formulate consciously. Extensive interviews and observations can identify enough of these rules to lead to selection forms that are competitive in performance with actual experts. An example set of rules is given in Section 4.B.

There is a large difference in perspective between mathematical problem solving and rule based selection forms. In the former one expects the exact answer or, at least, an answer extremely close to the exact answer (say, differing only in the sixth decimal place). In the latter one expects the best or correct answer perhaps only 70 or 80% of the time. This is the performance level of human experts and it is often unclear whether enough information and/or knowledge is available to allow significantly better performance.

4.1.3 Decision Trees. One way to systematize rules is to organize them into a decision tree, see the example in Section 4.B. Decision trees have been in use for a long time and their structure is easy to understand. At each node of the tree a value is computed using selection form parameters and problem features. This value is then used to select one of the branches from the node. Simple examples are:

Has the applicant filed for bankruptcy?:	Yes → left child node
	No → right child node
What is the applicant's home mortgage?:	<\$20K → child node #1
	\$20K – 50K → child node #2
	\$50K – 150K → child node #3
	>\$150K → child node #4

The computations at the nodes are traditionally simple, but there is no reason to exclude complex or lengthy computations. There may be numerous numerical parameters in a decision tree but, as for most forms, the structural parameters are the most important in determining potential performance. The structural parameters are the tree shape (number of levels, number of branches from each node) and the assignment of computations to the nodes.

A lesson learned from approximation theory may be paraphrased as follows:

If the tree structure is chosen poorly, then no amount of optimization of the numerical parameters will produce good performance.

If the tree structure is chosen well, then any reasonable way to choose the numerical parameters will produce good performance.

4.1.4 Neural Nets. The neural network is a selection form that accepts the problem properties at the top level nodes of a network, applies a simple rule at each node, and sends the result to all of the second level of network nodes. This may be reprcated with several levels, see the examples

in the Section 4.B. The key idea of the neural network is: *make the rule at each node independent of the problem.* There are only numerical values in the rules which are problem dependent. The classical node rule is:

Sum the node inputs, call it SUM
if SUM > *a then output 1*
else output 0

Thus there is one numerical parameter per node to be determined in this classical case. As before, the most important parameters of the neural network are structural, the number of levels of nodes, the choice of problem properties to use, and the rule used at the node.

The computation of 0 or 1 uses the step function (mathematically, $(\mathrm{SUM}-a)_+^0/|\mathrm{SUM}-a|$) and other functions have been used instead. For those problems with an enormous number of properties (e.g., image processing problems), one has a feature extraction phase before the top level of the neural network. Employing the neural network philosophy, this extraction is often very simplified.

4.1.5 Other Selection Forms. The selection form can be an arbitrary computation. In particular, it can be quite particular to the problem being solved and thus not fall into one of the above classes. One might use a simple table lookup. *Polyalgorithms* are used in numerical problem solving where one has a variety of numerical methods available with a number of logical tests determining when to switch between methods. These decision are quite dynamic, so one cannot predict the logical decision structure in advance.

4.2 Examples

Examples are given of selection forms using rule sets, decision trees and neural networks. Three neural network examples are given as this selection form is most closely related to optimization and approximation. These examples illustrate that while in theory selection forms may be any computable function, that in practice they tend to

(a) have numerous parameters which may be numerical, logical, symbolic or geometric,

(b) use numerous features from the problem,

(c) have a simple, repetitive nature in their structure.

Example 4.1. A Rule Set for a Simple Economic Advisor [6].

This rule set is for a book publisher executive who sets sales quotas for the national sales staff. There are 14 different types of books and the sales quota is for each of the four quarters of a year. Thus 56 quotas are needed for each salesman. The executive uses many variables (e.g., type of book, sales trends, advertising, past performance of salesman, season, territory) to adjust the quota.

R1 = IF: Sales > 1.15 * Quota
Then: Base = Quota + (Sales − 1.15 * Quota)

R2 = If: Sales <= 1.15 * Quota. Then: Base = Quota

R3 = If: Economy = "good" and Known ("Growth")
Then: Efactor = Growth

R4 = If: Economy = "fair" and Known ("Localads") and Known ("Growth")
Then: Efactor = Growth/3; Lafactor = Localads/120000

R5 = If: Economy = "poor" and Known ("Growth") and Known ("unemployment")
Then: Efactor = Min(Growth, .085 − Unemployment)

R6 = If: Growth >= .04 and Unemployment < .076
Then: Economy = "good"

R7 = If: Growth >= .02 and Growth < .04 and Unemployment < .055
Then: Economy = "good"

R8 = If: Growth >= .02 and Growth < .04 and Unemployment < .055 and
Unemployment < .082
Then: Economy = "good"

R9 = If: Growth < .02 or Unemployment >= .082
Then: Economy = "poor"

R10 = If: Economy = "good" and Localads > 2000
Then: Lafactor = Localads/100000

R11 = If: Economy = "poor" and Localads < 1500
Then: Lafactor = − .015

R12 = If: (Economy = "poor" and Localads >= 1500) or
(Economy = "good" and Localads > 2000) Then: Lafactor = 0

R13 = If: Prod in ["computer", "romance", "scifi"]
Then: Pfactor = (Newtitles + Oldtitles) / Oldtitles + 1
Strong = True; Weak = False

```
R14  =  If: Prod in ["reference", "biography", "psychology", "sports"]
        Then: Pfactor = .75 * ((Newtitles + Oldtitles) / Oldtitles − 1)
              Strong = False; Weak = False

R15  =  If: Strong and Known ("Base") and Known ("Efactor") and
            Known ("Pfactor") and Known ("lafactor")
        Then: Input Rise Num with "Enter estimate of % sales increase" \
              + "due to rising interest in " + Prod
              Newquota = Base * (1 + Efactor + Lafactor + \
                                 Pfactor + Rise/100)

R16  =  if: Weak and Known ("Base") and Known ("Efactor") and
            Known ("Pfactor") and Known ("Lafactor")
        Then: Input Fall Num with "Enter estimate of % sales decrease" \
              + "due to falling interest in" + Pod
              Newquota = Base * (1 + Efactor + Lafactor + \
                                 Pfactor − Fall/100)

R17  =  If: Not (Weak or Strong) and Known ("Base") and
            Known ("Efactor") and Known ("Pfactor") and Known ("Lafactor")
        Then: Newquota = Base * (1 + Efactor + Lafactor + Pfactor)

R18  =  If: Not (Prod in ["computer", "romance", "scifi", "reference",
            "Biography", "psychology", "sports"])
        Then: Weak = True; Strong = False
              Pfactor = .45 * ((Newtitles + Oldtitltes)/Oldtitles − 1)
```

Example 4.2. A Decision Tree for the Game of Checkers

Playing checkers is one of the nice accomplishments of artificial intelligence. In the 1960's A. Samuel used the game of checkers to study the possibilities for sophisticated machine learning, [13]. This work has matured into a computer program which is competitive with the very best international champions of checkers. This program progressed through a series of more and more specific selection forms until it reached the form illustrated in Figure 2. Its operation is as follows:

1. Compute features based on the particular game configuration. There are 24 of them, each having 3 or 5 values, placed into 6 groups. In each group there are 63 possible combinations of values (many of the potential 270 combinations are eliminated by symmetry).

2. For each combination of features, one of 5 possible values is computed at the first level of the tree. These output values are put into 2 groups with 125 possible combinations.

3. For each combination of level one outputs (in each group), one of 15 values is computed from a table.

4. There are 225 possible pairs of output from level 2 and a table with 225 entries is used to select the move.

This approach is further refined by identifying six phases of the game and having a separate decision tree for each phase. The determination of the phase is another feature computed from the game configuration, it depends primarily on the number of pieces on the board. There are 5118 table entries in these 6 decision trees, each of which is a parameter of the selection form.

Example 4.3. Basic Neural Network

The basic idea of the neural network appears as an imitation of processes thought to occur in the brain. Figure 3 illustrates how this imitation leads to the neural network. First, there is the complex of nodes (neurons), see Figure 3a, which combine and pass on information. Actual neuron nets in the brain have many more interconnections than illustrated here. The "combining" of a neuron is modeled by (see Figure 3b);

Add up the inputs
Produce an output by applying a filter function

The multiple connections of the neurons is modeled by a network of nodes starting with, usually, many inputs (problem properties) and a few outputs (solutions). In Figure 3c we see a small network with 10 inputs (say data on running, throwing, hitting of a baseball player) and 2 outputs (say, "draft" and "pass"). This might be used by a baseball team to select players for a draft.

Example 4.4. Identification of Underwater Objects

Figure 4 illustrates a typical military neural network application. One has a sonar operating underwater and the signal returned (the sound) has a power spectrum (plot of amplitude versus frequency of the sound) which has been smoothed. This is the curve shown at the bottom of Figure 4. This data is sampled at 60 points, only 9 are shown to simplify Figure 4. These data are the features that are extracted from the problem properties (the power spectrum). The network has 12 intermediate nodes (only 5 are shown) and two output nodes: "cylinder" (= submarine) and "rock".

This example is typical in that the network is broad (has many inputs), shallow (has few levels), and is highly interconnected.

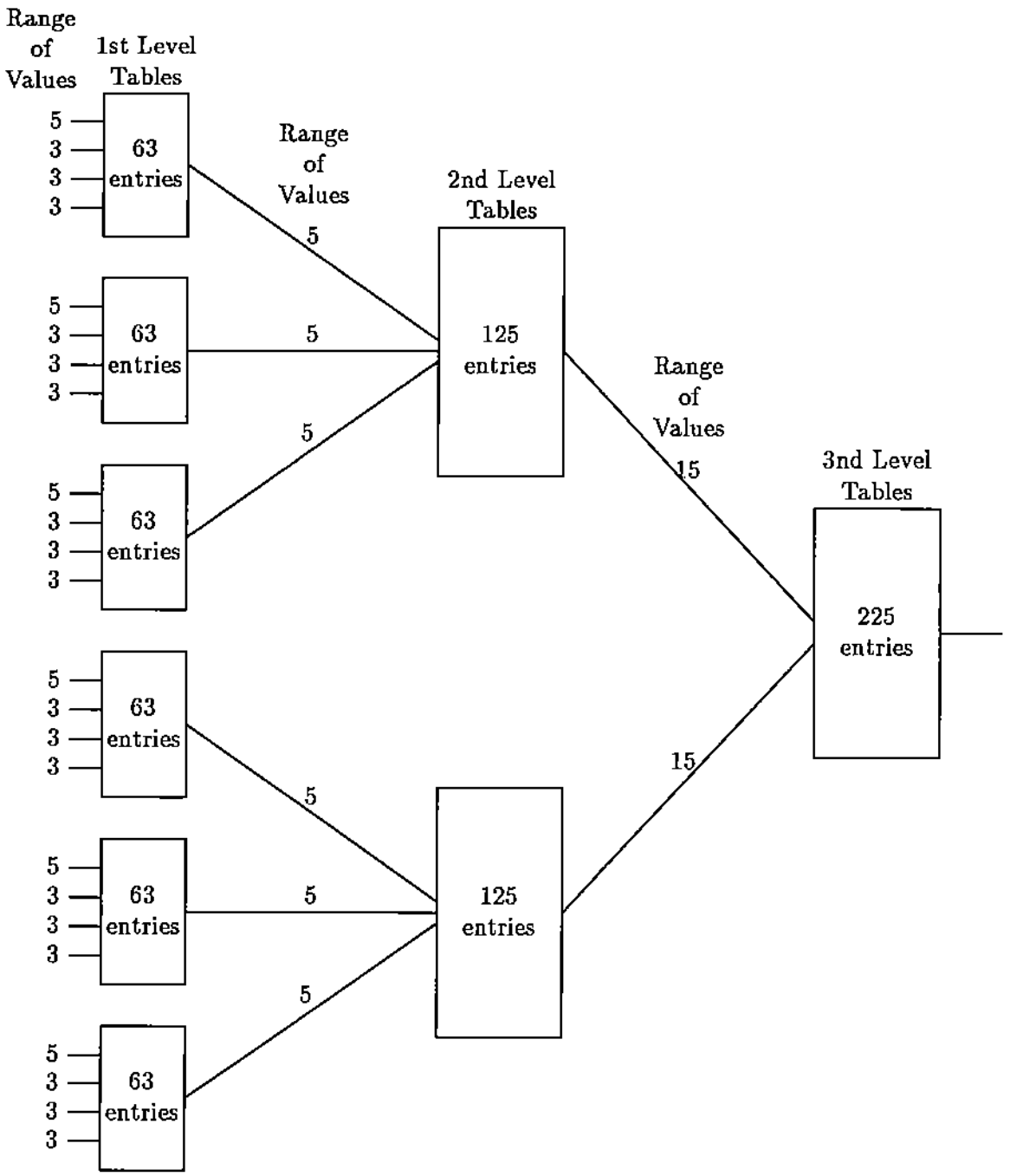

Figure 2: A decision tree for playing checkers. Six such tress are used, one for each phase of the game.

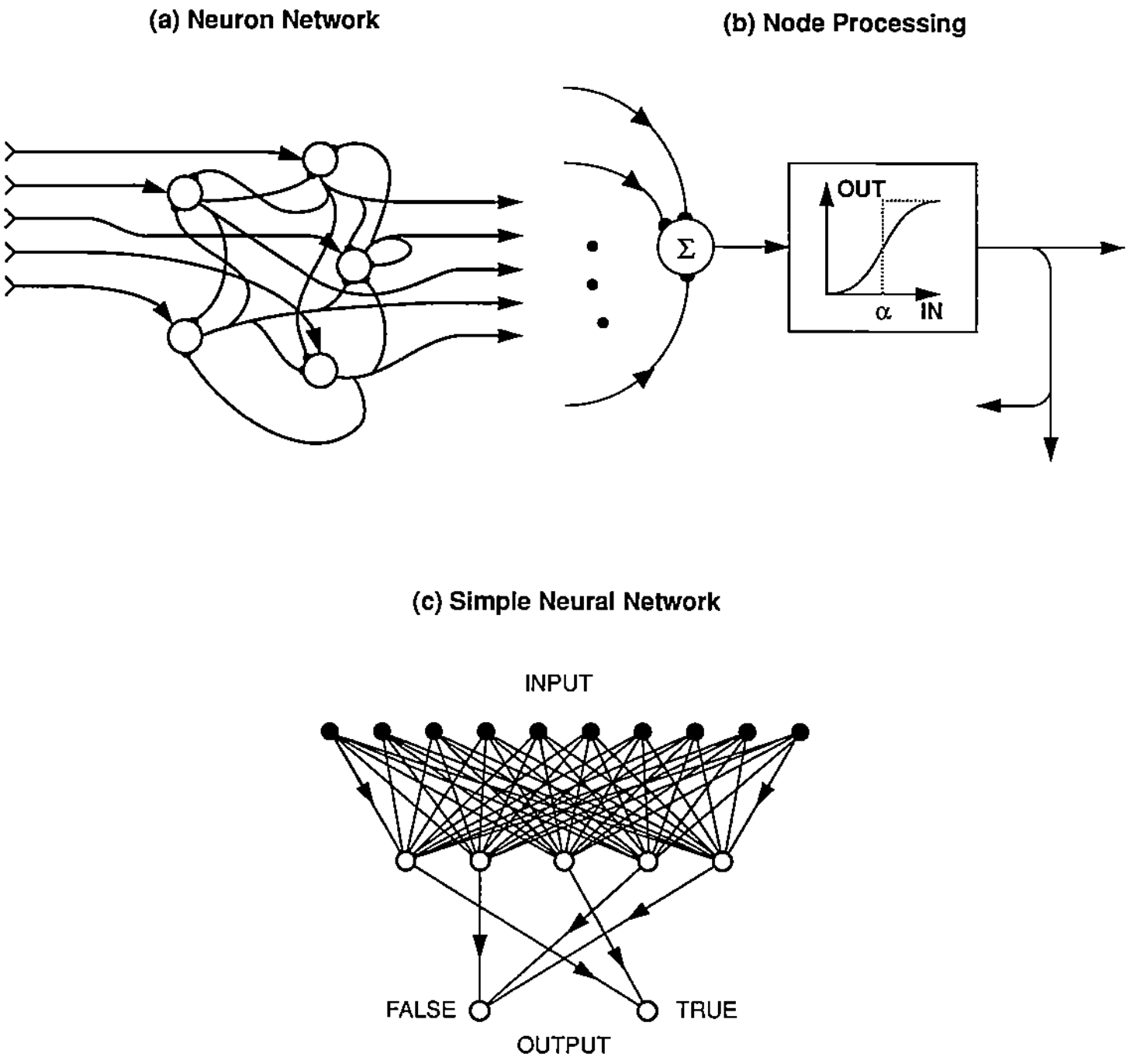

Figure 3: (a) Simplified schematic of networks of neurons in the brain. (b) Model of the processing done at each neuron or node. (c) A neural network to select two items based on 10 input values.

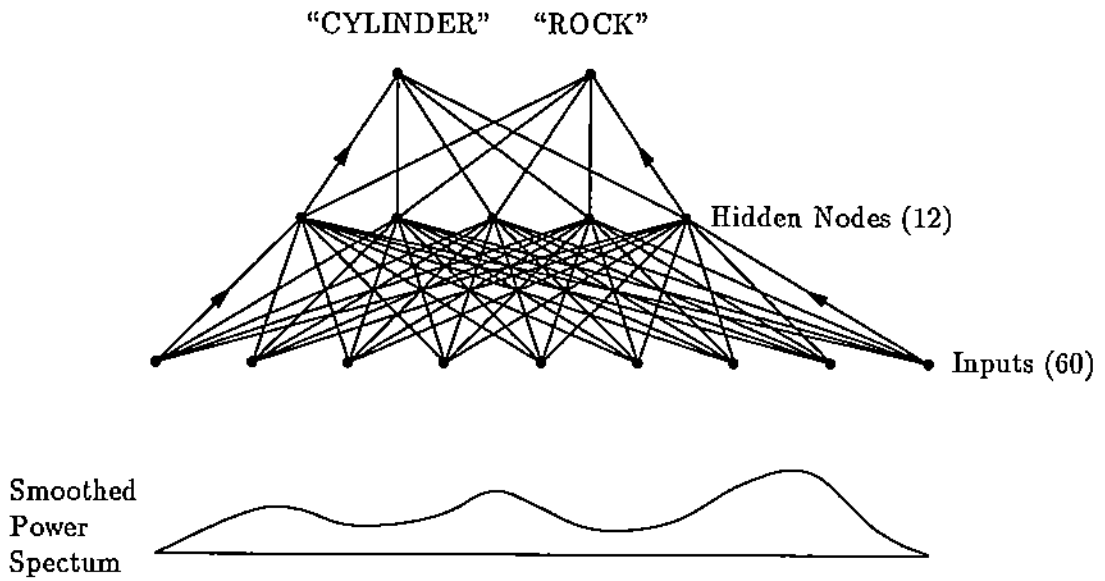

Figure 4: A neural network to identify underwater objects from sonar.

Example 4.5. Balancing a Broomstick

A classical control problem that every child learns to solve is how to balance a long stick on a finger. One soon learns how to move the base of the stick to maintain balance while watching the stick. A version of this problem is illustrated in Figure 5 where the stick is placed on a little car and pivots freely there. The problem properties (the view of the car and stick shown on the left) are replaced by a 5 by 11 grid. A grid is marked black or white depending on whether the car and stick are either present or absent in the grid. A threshold is used to measure the portion of the grid which must contain the car and stick before the grid is black.

A neural net with three levels is created with these 55 problem features as inputs and with two outputs "accelerate left" and "accelerate right". After sufficient "training", the parameters of the neural net were determined so a computer could keep the stick (appropriately simulated) upright for long periods.

This is an example where cleverness would probably greatly simplify this control task. From the problem properties it is straightforward to compute two features, namely:

the angle of tilt of the stick,
the relative velocity of the top and bottom of the stick.

We suspect that a neural net using these features would not only be simpler, but it would also be much simpler to determine good parameters.

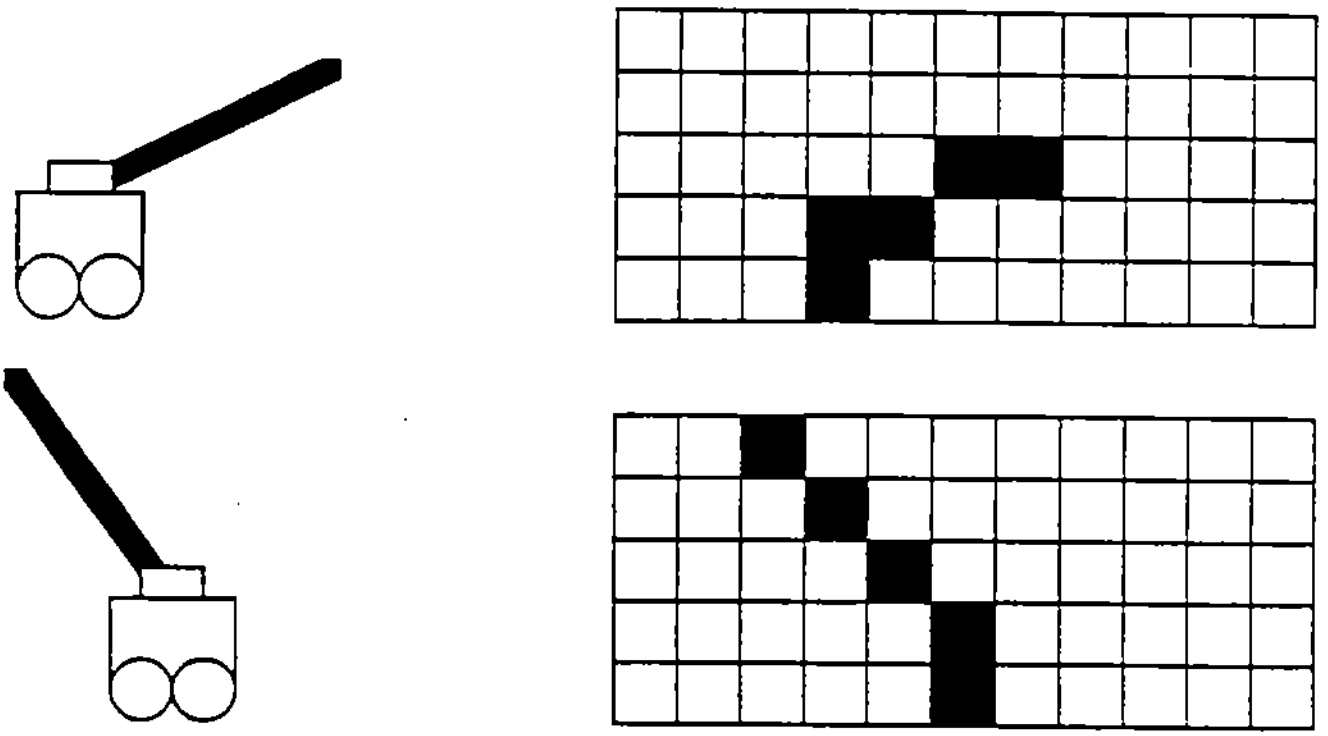

Figure 5: Two views (a) of the "Balance a Broomstick" problem and the corresponding features (b) used as input for a neural net to control the car's motion.

Example 4.6. Choosing a Method to Solve Elliptic Problems

The ELLPACK system has a large database of elliptic problems and the performance of many methods to solve these problems. For some problems all applicable methods have been applied. The Athena project [ref] is an exemplar based approach to using this database in order to learn how to choose good methods for solving elliptic problems.

The approach is as follows. One identifies a set of exemplars (example problems) where enough data is available for knowing or estimating well the best method to use for the exemplar. Given a new problem, one locates the exemplar nearest to it as measured by the 36 problem features. It clearly takes an enormous number of exemplars to populate a 36 dimensional space even sparsely. However, there are two reasons to hope for this approach to succeed with only a few hundred or thousand exemplars. First, there is considerable structure in these features, some are mutually exclusive of others, so the space of features is actually of much lower dimension than 36. Second, there is considerable hope – and some evidence – that many of the features are "orthogonal" to one another; that is, there is little interaction between these features in determining which methods to choose. Thus, it remains to be seen how many exemplars are actually needed for good learning performance. Figure 6 shows a set of performance data and extrapolating lines for one aspect of solving one elliptic problem [12].

Note that even if only a modest number of exemplars are required, the database will be very large. Just to determine the performance of all the interesting methods for one elliptic problem requires using many different mesh sizes and combinations of PDE solving components. Thus, one could easily require solving a thousand elliptic problems to generate the performance information needed to choose a method for just one exemplar. Note that one needs performance data for methods that are not the best because a new problem could be close to an exemplar and have properties that preclude some methods from being applied. Thus one needs data about a variety of methods for each exemplar.

5 What Are Learning and Teaching?

We identify three distinct phases to learning how to solve a problem:

Phase 1: Construct a Selection Form. This is the most important and most difficult part of problem solving. It is also the hardest to analyze. Much of the progress in automated problem solving (a branch of machine learning) has been due to the discovery and wide use of selection forms which are widely applicable and very flexible.

Phase 2: Identification of Problem Features. Next most important is to chose an appropriate set of problem features as input variables to the selection form. Naturally, there is considerable interplay between Phases 1 and 2, but for some generic selection forms, these two phases are well separated. The identification of features (as opposed to the computation of appropriate features once identified) is well known to be one of the difficult challenges of problem solving.

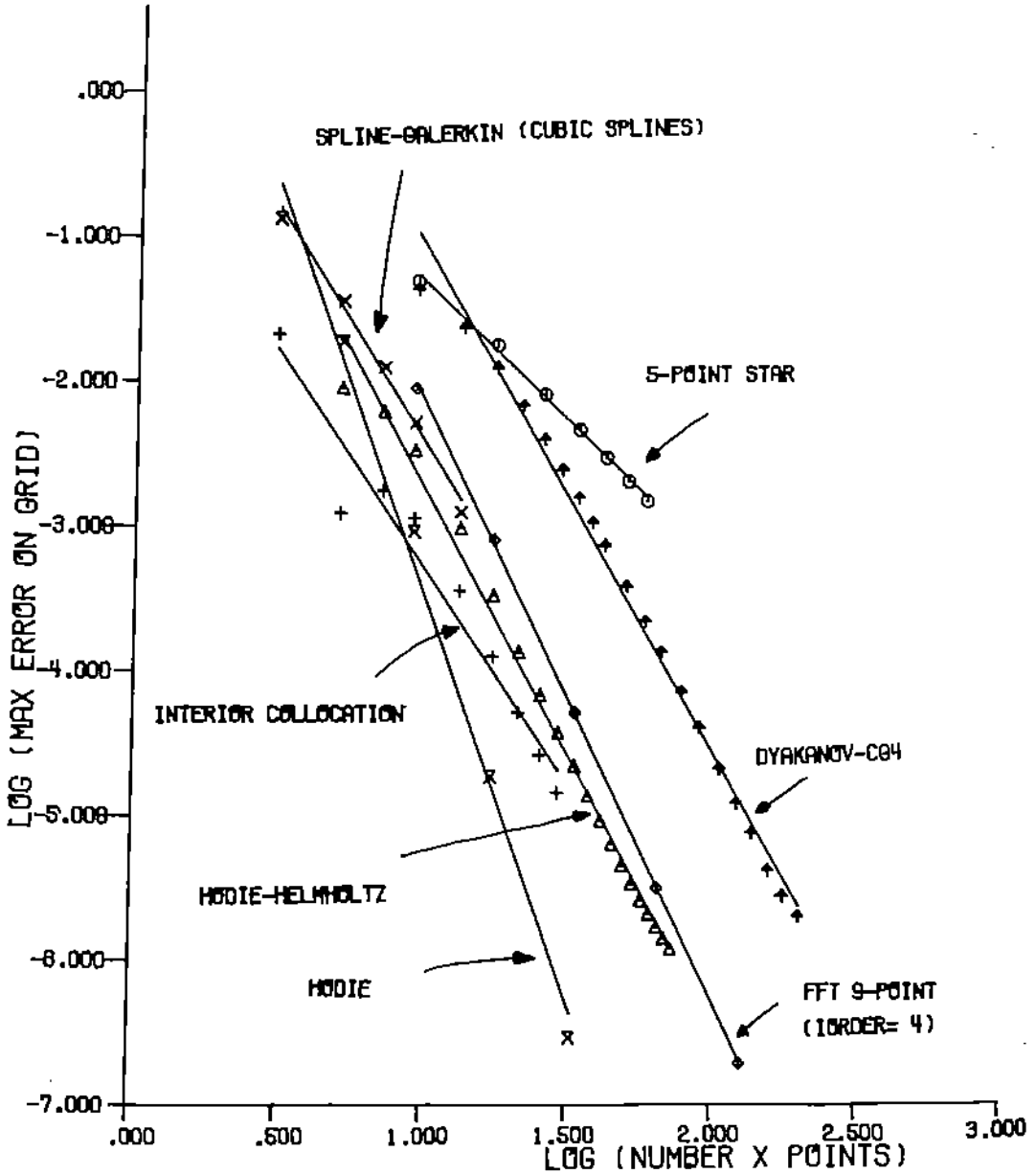

Figure 6: Performance curves of computer time versus accuracy requirement for the elliptic problem (exemplar) $u_{xx} + u_{yy} = f(x, y)$ with Dirichlet boundary conditions on a rectangular domain.

Phase 3: Optimization of the Parameters of the Selection Form. Once the form and features are chosen, then there is normally a substantial number of parameters to determine which control the success of the problem solving. If these parameters are numerical (as they often are, including yes–no logical parameters) then computer science can bring its analytical tools and supercomputers to bear on learning how to solve a particular problem. The task has been reduced to an optimization problem: Determine parameter values to maximize the "grade" (see Figure 1) of the solutions obtained.

With these three phases to learning, we see that the teaching is the exhibition of good, even optimum, solutions to instances of problems. One might say that the teacher solves problems while the student observes. However, there are other purposes and activities of teaching, namely,

a. Identification of useful selection forms. The student sees the selection forms that the teacher uses and gains an appreciation for their effectiveness. This identification might be very explicit in the case of mathematical forms used in analytical methods. The teacher may use rules in the problem solution so the student can learn specific rules as well to build up a collection of rules as a base for future rule creation. This collection is just one example of the student learning to make abstractions, to identify common patterns in problem solving.

b. Identifications of useful parameters and features. Experience in problem solving allows the student to create a taxonomy of features relevant for certain problem classes. The teacher might even explicitly describe the taxonomy and illustrate how to use various types of features. Similarly, the student sees the parameters used in problem solving and learns appropriate values for many of them. Finally, as above, the student builds a base of information from which abstractions can be made and, later, new features created for new, but similar, problems.

c. Practice in executing selection forms. In a great deal of problem solving the algorithms are implicit, unseen by both the teacher and student. An example is the skill of indefinite integration learned in calculus courses. It is now known that there is an algorithm that solves this class of problems but no one learns it. Rather, each student builds his own algorithm (which often performs poorly) by observing the teacher and by *practicing its application.* Thus the practice (i.e., homework) aspect of teaching serves both the purpose of having the student execute his algorithm for efficiency and to test its correctness. When it is incorrect the teacher shows the student the relevant problem solutions so the algorithm can be modified. All of this practice builds the base for future problem solving.

6 Where is the Knowledge?

The focus here on solving specific problems obscures the relationship between knowledge and learning. The learning described here results in a large collection of computational methods (programs). The selection forms, features, parameters values, etc., of these methods form the knowledge about solving the problems. We do not see in the present discussion the large array of facts often identified as knowledge, for example, the information contained in the Encyclopedia Britannica. However,

we do have large arrays of information about problem solving and an experienced problem solver would probably fill many volumes if all the details of his methods were recorded.

Is knowledge then a large set of computational methods to solve problems plus a large database of facts? There is surely one other component, an organization structure relating all these things together. One can view the selection forms and features as imposing an organizational structure on the problem properties. This structure is special purpose, its goal is to solve a particular problem. There should also be structures that relate general items together in interesting and useful ways. For example, the abstractions that one creates during learning are of this nature.

7 Lessons From Optimization Theory

7.1 Summary

Optimization is primarily concerned with determining the best value for some purpose. This is formalized as follows: We have an objective function $f(x, p)$ where x is a variable to be varied and p represents problem data which is fixed. The function f evaluates to a number and we assume that x is to minimize this function. Then optimization is to

$$\textit{minimize } f(x,p) \textit{ as a function of } x$$

It is often convenient to identify constraints that must be satisfied by x. In principle these can be incorporated into f but in practice is often easier to keep them separated so the optimization problem is then restated as

$$\begin{array}{l}\textit{minimize } f(x,p) \textit{ as a function of } x \\ \textit{subject to the constraints } c(x,p) = 0.\end{array}$$

Note that the theory and practice of optimization is very closely related to that of solving nonlinear equations. The value of x that minimizes $f(x, p)$ is one which solves $df(x, p)/dx = 0$.

Traditional optimization involves various standard mathematical functions for $f(x, p)$ and the constraints. For example, the problem

$$\begin{array}{ll}\textit{minimize} & p_1 + p_2 x + p_3 e^{-p_4 x} \\ \textit{with} & p_2 x + p_4 x^3 - p_5 \geq 0 \\ & p_6 x + p_7 x^2 - p_8 \leq 0\end{array}$$

has eight data values p_i, $i = 1, \ldots, 8$ from the problem. It is easy, but not helpful, to convert the pair of inequality constraints here into a single equality $c(x, p) = 0$.

Optimization theory and practice has generated many ideas and methods. Some are very general, e.g., gradient descent, convex minimization, penalty functions for constraints. These can be and have been used in the determination of best – or good – values for the parameters of selection

forms. There are many special problem classes identified in optimization theory [9] and one of the important lessons learned is

> *Exploitation of the special nature of a class of problems can have enormous benefits in increasing the reliability and efficiency of optimization methods.*

This lesson seems quite obvious but it has important consequences. For example, it suggests that general purpose optimization methods should only be used when

a) the problem at hand is quickly solved by one of them,

b) the cost of analyzing the problem at hand exceeds the cost of using an inefficient method.

c) analysis of the problem has yielded no insight into the creation of better optimization methods.

A second lesson learned is

> *There are no generally reliable methods for optimization.*

This can be seen by performance evaluations [5] of software packages for optimization and solving nonlinear equations. These evaluations involve a set of 100 or so problems to solved to which each software package is applied. The results are, roughly,

a) every package fails on 10 to 25% of the test problems,

b) every problem is solved by some software package.

A third lesson learned is

> *It is irrational to expect to be able to solve even well behaved optimization problems involving many variables unless they have some very special structure.*

By "many variables" we mean that x is a vector of 10 or 100 or 1000 variables. By "very special structure" we mean that $f(x,p)$ is convex or linear or something similar. By "well behaved" we mean that $f(x,p)$ is smooth and any cross section (obtained by restricting the components of x to a line) is simple, e.g., has at most two local minimia. Note that if every cross section has only one local minimum, then f is convex.

In the next section we show by example why this lesson is true. This lesson raises a very interesting question: *Why is it that we can solve so many problems involving many variables when it should be impossible to do so?* We regularly see solutions of optimization problems with thousands of variables, problems which are not convex, not linear and which seem to have no special properties. This implies that there frequently are special structures in real world problems which we do not, as yet, recognize. It would seem very important to determine what these structures are.

7.2 Example: High Dimensional Optimization is Impossible

The two examples presented here are derived from the observation that the bulk of the volume of a high dimensional cube is in the corners of the cube. We define the N-cube as the set $\mathbf{x} = (x_1, x_2, \ldots, x_N)$ with $-1 \le x_i \le +1$. Its volume is 2^N and its diameter is $2\sqrt{N}$. The N-sphere in the set $\mathbf{x}$ with $\sum x_i^2 \le 1$ and its volume is $2^{N-1}\pi/N$. The ratio of the volumes of the N-cube to the N-sphere grows unboundedly with N. Suppose that $f(x)$ has a dip (local minimum) in each corner of the N-cube and outside the N-sphere. Then $f(x)$ has 2^N local minima and the only way to find the lowest one is to compute them all as they can be entirely separated from one another. Furthermore, if the dips are located near the N-cube vertices, then no line can pan through more than two of them. Two simple, explicit examples are given of such functions. Clearly no "fast" algorithm can minimize these two functions numerically with arithmetic work of order less than 2^N. For the second example it is doubtful that even examining the symbolic definitions of $f(x, p)$ can lead to the global minimum with work of order less than 2^N.

Example 7.1. The Pocket Function.

This function $y = pocket(x)$ was created by Carl deBoor using just three lines of MATLAB [10] code.

```
depth  =  cos(sum (( 1 + size(x)). * cumprod(2 * ones (x))) − 2;
point  =  min([norm(abs(x) − ones(x)/2) * 2, 1]);
    y  =  depth * (1 − point * point * (3 − 2 * point ));
```

In traditional mathematical notation this function is defined as follows: Given x compute:

$$\begin{aligned} depth &= \cos[\textstyle\sum_{i=1}^{N} 2^i(1+\mathrm{sign}(x_i))] - 2, \qquad -3 \le depth \le -1 \\ p &= \mathrm{dist}(|\mathbf{x}|, (1/2, 1/2, \ldots, 1/2) \\ pocket &= depth * (1 - p^2(3 - 2p)) + 1 \end{aligned}$$

This function has a "pocket" in each corner of the cube $|x_i| \le 1$ of depth given by the first expression. The lowest point in each corner is at $\mathbf{y}$ where $|y_i| = 1/2$ for all i and the variation about $\mathbf{y}$ is a cubic polynomial with minimum at $\mathbf{y}$ rising to a value of 0 (with slope = 0 also) at a distance of 1 from $\mathbf{y}$. The minimum of $pocket(x)$ can be found by evaluating the 2^N cosines that determine *depth*. This function is defined also by the following MATLAB code which allows simultaneous evaluations of $pocket(x)$ for several x.

```
[d,n]  =  size(x);
depth  =  cos(sum((1 + sign(x)) . *(cumprod(2 * ones (d,1)) * ones (1,n)))) − 2;
point  =  min([sqrt(sum((abs(x) − (ones(d,1)/2) * ones(1,n)). ^ 2))*2; ones(1,n)]);
    y  =  depth.*(1−point.*point.*(3−2*point));
```

Example 7.2. The Pocket-p Function

This function $f(\mathbf{x}, p)$ has a real parameter p in order to create a large set of functions similar to the pocket function and to avoid having a global minimum that can be computed a priori from the symbolic definition of f.

Parameter p. We construct a set of 2^N numbers from the binary expansion of p as follows:

$$\begin{array}{rcll} \alpha_i & = & p \bmod(2^{-i}) * 2^i & O \le \alpha_i \le 1, i = 1, 2, \ldots, 2^N \\ \beta_i & = & p \bmod(2^{-(i+3)}) * 2^i & O \le \beta_i \le 1/8 \\ \delta_i & = & p \bmod(2^{-(i+6)}) * 2^{i+6} & O \le \delta \le 1 \end{array}$$

We let $\mathbf{v} = (v_1, v_2, \ldots, v_{2N})$ be a vector of 0's and 1's formed from the binary digits of p. We define $p \bmod(2^{-i})$ as the number obtained by setting to zero all the binary digits of p from 1 to i. Thus if $p = 3.40625 = 11.01101$ then $p \bmod(2^{-2}) = 00.00101 = .15625$. These numbers are fixed for $f(\mathbf{x}, p)$.

Definition of $f(x, p)$. Given $\mathbf{x} = x_i, x_2, \ldots, x_N$ with $|x_i| \le 1$ compute:

$$\begin{array}{rcl} \mathbf{s} & = & (\text{sign}(x_1), \text{sign}(x_2), \ldots, \text{sign}(x_N)) \\ k & = & 1 + \sum_{j=1}^{N} (1 - s_j) 2^{j-2} \end{array}$$

The index k is between 1 and 2^N and is used to identify the corner containing $\mathbf{x}$. Compute

$$\begin{array}{rcl} \mathbf{c}_k & = & 3\mathbf{s}/4 \\ r_k & = & 1/(4 + \alpha_k) \\ \tau & = & \|\mathbf{x} - \mathbf{c}_k\| \end{array}$$

IF $\tau \ge r_k$ then

$$f(\mathbf{x}, p) = 1$$

else

$$f(\mathbf{x}, p) = [1 - \delta_k \cos(2\pi\tau(4 + \alpha_k))] + \beta_k \; \delta_k \; p_5(\mathbf{x})$$

endif

The quintic polynomial $p_5(\mathbf{x})$ is computed as follows. Determine $\mathbf{w}$ as a vector of length N by $w_i = v_{k+i-1}$. Set

$$
\begin{aligned}
\gamma_k &= \mathbf{c}_k + \beta_k/\sqrt{N} \\
d &= \|\mathbf{c}_k - \gamma_k\| \qquad\qquad 0 \le d \le 1/8 \\
t &= \|\mathbf{x} - \gamma_k\| \\
y &= (\mathbf{x} - \gamma_k)/t \\
\cos\theta &= (\gamma_k - \mathbf{c}_k)\cdot \mathbf{y}/d \\
g &= d\cos\theta \\
q &= g + t \\
h^2 &= d^2(1-\cos^2\theta) \\
\ell &= \sqrt{r_k^2 - h^2} \\
p_5(\mathbf{x}) &= \frac{(\ell^2-q^2)^2}{\ell^2-g^2}\,[1 - \frac{2g(g-q)}{\ell^2-q^2}\,]
\end{aligned}
$$

The polynomial p_5 has the following properties: On the line through γ_k and $\mathbf{x}$ it is a quintic polynomial which satisfies

1) p_5 and $p_5' = 0$ at the boundary of the sphere of radius r_k centered at $\mathbf{c}_k$.

2) $p_5 = 1$ at γ_k and $p_5' = 0$ near γ_k.

The purpose of this quintic polynomial is to perturb the local minimum of $f(\mathbf{x}, p)$ slightly away from $\mathbf{c}_k$ in a somewhat unpredictable way. That is, one must carry out a local minimization of $f(\mathbf{x}, p)$ near $\mathbf{c}_k$ in order to determine the value of $f(\mathbf{x}, p)$ at the local minimum.

8 Lessons From Approximation Theory

8.1 Summary

Classical approximation theory is concerned with approximating a given function, say $g(t)$, by a combination of others, say

$$f(\mathbf{x}, t) = \sum_{i=1}^{k} x_i\, b_i(t)$$

where the $b_i(t)$ are basis functions. This is linear approximation since the x_i enter linearly , there is a corresponding non-linear approximation problem which may be expressed

$$\text{minimize}_{\mathbf{x}} \quad \|g(t) - f(\mathbf{x}, t)\|$$

where $\|\cdot\|$ is a norm that measures the distance between functions. Approximation theory addresses two principal issues:

a) *How well can the approximation be done?*

b) *How to compute best (or good) parameters* x *of an approximation.*

The second issue is a special case of optimization and there is a strong relationship between computational methods in approximation and in optimization. The effort in computational approximation theory is to exploit special properties of the problem in order to obtain better efficiency.

The issue of how well the approximation can be done is related to choosing a selection form for learning. Thus choosing different basis functions, e.g.,

polynomials:	$1,\ t,\ t^2, \ldots t^k$
trigonometric:	$1,\ \sin(t),\ \cos(t),\ \sin(2t),\ \cos(2t),\ \ldots$
exponentials:	$1,\ e^t,\ e^{-t},\ e^{2t}, e^{-2t}, \ldots$

is equivalent to choosing different selection forms.

Some of the lessons learned from approximation theory are:

a) *Choosing an appropriate mathematical form for* $f(\mathbf{x}, t)$ *is essential to obtaining good approximations.*

This lesson has already been paraphrased at the end of Section 4.13. Another instance of this is: If $\delta(t)$ is oscillatory, then trigonometric basis functions are likely to approximate well while polynomials are unlikely to approximate well.

b) *The fact that one can prove that a mathematical form can always approximate well does not imply that this form is so useful in practice.*

The famous Weierstrass Theorem states that given any continuous function $g(t)$ and any accuracy criterion $\in > 0$, then there is a polynomial $p(\mathbf{x}, t)$ so that

$$||g(t) - p(\mathbf{x}, t)|| \leq \in$$

This suggests that polynomials should be adequate for any approximation problem. In fact, however, the polynomial degree required is frequently very high and the required polynomial approximation is too difficult to compute and too difficult to use.

A true breakthrough occurred in approximation theory in the 1960's: the application of piecewise polynomial and spline methods to one dimensional approximation. An episode of this breakthrough is given in the following example. These mathematical forms are defined in terms of a set of *knots* or *breakpoints* $T = t_0, t_1, \ldots, t_K$ and the function $f(\mathbf{x}, t)$ is a polynomial of degree n between these knots. Further, the polynomial pieces are constrained to join up smoothly at the knots. The smoothness might vary from simple continuity to $n/2$ continuous derivatives (a common choice) to

$n-1$ derivatives (the maximum possible smoothness, giving splines). Both the knot locations and the polynomial coefficients are parameters to be determined in achieving a good approximation. As with ordinary polynomials, one can prove that any function $g(t)$ can be approximated well by piecewise polynomials or splines. Unlike polynomials, this can be achieved in practice as well as theory.

These approximating functions provide an actual case where a general purpose, flexible selection form has been found: To obtain a good approximation of a function of one variable, one merely need choose the piecewise polynomials and apply known algorithms to compute the parameters. In fact, algorithms exist which also compute the number of pieces needed as well. Extension of this idea to approximating functions of 2 or 3 variables (e.g., surfaces) have been successful, but the methodology is more difficult and less robust. Thus we conclude with a third lesson:

c) *There is hope to find a general purpose approximation so that all that needs to be done is to compute good parameters values.*

8.2 Example: A Breakthrough of the 1960's

In the early 1960's a number of people began to appreciate the potential of piecewise polynomials and splines for general purpose curve fitting. The ground work had been laid in the late 1940's and 1950's but "production" use had not yet occurred. A group at the General Motors Research Labs began working on using splines to represent automobile body surfaces, surfaces where classical polynomial, trigonometric, etc., based methods did poorly. The people involved were Garrett Birkhoff, Carl deBoor, Hermann Burchard, Henry Garabedian and John Rice.

A program had been written to do variable knot, cubic spline, least squares approximation. A technical writer was in the department one day asking about the work and was told that "we can compute a good smooth approximation to any shape". Two days later he returned with a photograph of his secretary, shown in Figure 7, and asked that we fit the girl's profile.

No one, to our knowledge, had ever approximated such a complicated curve by a smooth mathematical form. The photograph outline was discretized to 152 points, not uniformly spaced, and a slight cheating was made by not including the eyelashes. The program was set to work on the problem and within a few days an accurate cubic spline approximation was found with 18 polynomial pieces. Recall that in those days, one could only expect one or two runs of a program in a 24 hour period and multiple runs were needed as the program would only adjust the knot locations and not the number of pieces. The approximation shown in Figure 7 only has 16 pieces, it was computed a few weeks later. This approximation is as accurate as the original data (obtained from tracing paper laid over the photograph), it is smooth (having two continuous derivatives), and it has the right "shape" everywhere (including at the mouth which is particularly difficult for a smooth curve).

After this computation the group could, with confidence, claim that they could fit any curve.

Figure 7: Photograph (left) of a girl and an approximation (right) to the profile. The approximation is a cubic spline with 16 polynomial pieces computed in 1963 at the General Motors Research Laboratories.

9 Could There Be Magic Forthcoming?

We revisit the questions of learning to solve problems and formulate it from the approximation theory point of point. We then discuss the hope that supercomputers, superoptimizers and new selection forms can create solution methods "automatically" for broad classes of problems. We then discuss the difficulties we foresee.

9.1 The Hope

Problem solving is illustrated in Figure 8 where two paths are shown from the problem data to its solution. The left one is the "best" path, it associates problem instances with the best solution. How this solution is found is unknown, just as when we have a function $g(t)$, we need not know how it is computed. We merely have a $g(t)$ procedure that produces a value of g given a value of t. The right path is the "algorithmic" path, it is the one we wish to create, i.e., the one we will use once we have learned how to solve the problem. That path shows the three phases of problem solving identified earlier.

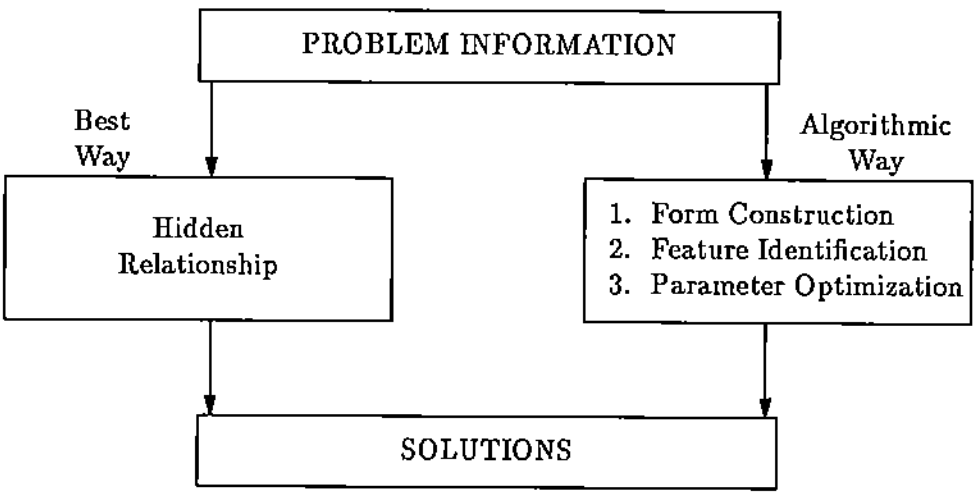

Figure 8: Learning to solve problems viewed as determining an algorithm (right path) to approximate the function (left path) that associates solutions with problems.

The hope is that there is a general selection form which, with a reasonable set of features and optimized parameters, will be able to approximate well the function $g(t)$ of the hidden relationship between problem information and solution. The hope further is that good parameters can be computed in reasonable time with commonplace high performance computers. This would extend the breakthrough in one-dimensional approximation that splines achieved in the 1960's. The examples given in Section 3 suggest some candidates for general selection forms. In recent years new op-

timization methods have appeared (e.g., genetic algorithms, simulated annealing) to complement the classical ones; these might be the superoptimizers we need. Finally, supercomputer power is growing rapidly, within a few years computers with 50-500 BIPS (Billion Instructions Per Second) will be in use and they will be widely available not long after that.

One can hope that this combination will provide automatic learning of problem solving for a broad class of problems, this could be the magic that machine intelligence research has been trying to develop.

9.2 The Difficulties

We briefly discuss four important difficulties in having this hope come true. The first two of these are somewhat unrelated to selection forms, computer power, etc. They are simply fundamental limitations on learning of all types, but it seems appropriate to mention them.

Performance Function Missing. There are many problems where very few, perhaps no one, know the quality of a proposed solution. Examples of this occur in playing chess, managing a company, or conducting a military campaign. There are many "moves" in these problems which can be evaluated as poor, but there are very few masters (and they are likely to disagree) available to teach a learning program.

No Problem Properties Available. There are problems that one can identify but not realize in a way to measure their properties. Examples of this are: (a) defending against an attach by alien space invaders, (b) controlling the "Star Wars" defense system, (c) managing saturated operations of a new computer system, and (d) handicapping the 1999 Kentucky Derby. The problem information is not available for these problems and there is considerable doubt that any abstract models of them will provide accurate problem information.

One can summarize these first two difficulties as one might either have all the problem information and not know what to do or one might not even have all the problem information. The next difficulty discussed is directly related to the hoped for automatic problem solving methodology presented in this section. This one appears to be a real limitation on the potential of this methodology.

Uncontrolled Complexity Growth. Once one has a generic approach to a complex problem, there is a tendency to throw in all the potentially useful information and let the method sort out which is actually important. The two adverse consequences of this tendency are: (1) the number of variables becomes very large, and (2) many of the variables become redundant or highly correlated (e.g., the speed, the wind resistance, the span, the fuel consumption rate, the instantaneous gas mileage of a car). The example in Section 8.2 shows that the work to optimize parameters can grow very rapidly with the number of parameters even for a simple, smooth function. None of the old or new optimization methods can cope with complexity of this sort. While the nature of the optimization problem being generated by automatic learning is not clear, it is certainly risky to assume that optimizations involving hundreds or thousands of parameters will be done quickly.

The use of redundant and highly correlated variables not only unnecessarily increases the com-

plexity of the selection form but it also can make the optimization problem much harder to solve. The dependencies between variables and their associated parameters make the computations ill-conditioned and much longer. In summary, it is likely that the complexity of the selection forms must be controlled (minimized) to a considerable extent. Thus we conclude that truly automated learning for problem solving must include ways to reduce the complexity of selection form, i.e., identify important variables (problem features) and modify/improve selection forms.

Hard to Algorithmically Identify Important Features and Modify Selection Forms. As stated before, the hardest part of this problem solving approach is to identify the right features and selection forms. A consequence of the arguments made above is that this part of problem solving cannot yet be completely replaced by supercomputers and superoptimizers.

10 Two Observations

10.1 The Complexity of Geometric Features

It appears that one of the difficult aspects of problem solving is handling geometric information well. The example in Figure 5 of *balancing a broomstick* uses a very crude way to reduce geometric information to numerical information. There are surely much better approaches than this. This situation reflects the fact that geometrical computing is in its very, very early stages compared to numerical and symbolic computing.

Methodologies are needed to algorithmically identify and refine important geometric variables. Vision is, of course, heavily involved in this activity and one only need examine that field to see how difficult this is. Yet for problem solving a slightly different methodology may also be needed, one that identifies geometric features that might not be visible objects but rather domains important to solving a particular problem. Some success has been achieved in numerical computation in algorithmically identifying geometric domains by techniques such as adaptive quadrature, adaptive curve fitting, adaptive mesh refinement and moving mesh methods for partial differential equations. The idea of these methods might lead to more generally applicable methods.

The idea of these adaptive numerical methods is as follows:

1. Define a simple geometric structure, with parameters, that covers the entire domain of interest. For example:

   Line: Divide it into 16 intervals.

   Rectangle: Divide into 10 by 10 mesh of quadrilaterals.

2. Fix the size of the structure (number of intervals or quadrilaterals, number of levels of refinement)

3. Define a **local** performance function on the whole domain.

4. Use this function to optimize the parameters of the geometric structure.

10.2 Automation of Rule Set Creation

The current practice in creating rule sets is to ask experts to suggest rules and then to optimize their parameters by monitoring their performance. This practice is related to problem solving as defined in this paper but choosing the selection form and identifying the problem features are merged together. Often, there is little effort in optimizing the parameters systematically. Rather, effort is expended in augmenting the rule set or changing the rules in more fundamental ways.

There is an aspect of this practice which is widely understood but rarely discussed in textbooks, etc. That is: *the principal difficulty in optimization is getting a ball park estimate of the optimum parameters.* This practice of using experts is good in the sense of obtaining good estimates. There is also a caveat to this practice, namely, *different experts are likely to give quite different opinions (rules) for the same situation.*

Open questions about the automated creation of rule sets include:

- How does one best effectively apply the ideas of optimization here?
- Should the rule processing system be modified to assist with parameter optimization?
- Would more structure in the rule sets make optimization and execution easier?

Acknowledgement

This work was supported in part by the National Science Foundation by its Institutional Infrastructure grant CCR-8619817.

References

[1] W.F. Ames, *Numerical Methods for Partial Differential Equations*, Second Edition, Academic Press, New York, 1977.

[2] Wayne R. Dyksen, and Carl R. Gritter, Elliptic expert: An expert system for elliptic partial differential equations, in *Intelligent Mathematical Software Systems*, (Houstis, Rice and Vishnevetsky, eds.), North-Holland, Amsterdam, (1990), pp. 23–32.

[3] T.F. Elbert, *Estimation and Control of Systems*, van Nostrand Reinhold, New York, 1984.

[4] Carl R. Gritter, *An Expert System for Elliptic Problems*, Ph.D. Thesis, Purdue University, (1992).

[5] K.L. Hiebert, An evaluation of mathematical software that solves systems of nonlinear equations, *ACM Trans. Math. Software*, 8 (1982), pp. 5–20.

[6] C.W. Holsapple and A.B. Winston, *Manager's Guide to Expert Systems Using Guru*, Dow-Jones Irwin, 1986.

[7] E.N. Houstis, J.R. Rice, A. Hadjidimos, and E.A. Vavalis, //ELLPACK: A numerical simulation programming environment for parallel MIMD machines, in *Supercomputing '90* (J. Sopka, ed.) ACM Press, New York, (1990), pp. 96–107.

[8] Elias N. Houstis, John R. Rice, and P. Varadoglou, Athena: A knowledge base system for //ELLPACK, (1991), to appear.

[9] D.G. Luenberger, *Linear and Nonlinear Programming*, Addison Wesley, Reading, MA, 1984.

[10] MATLAB, The Math Works, South Natick, MA, 1990.

[11] John R. Rice, , The algorithm selection problem, in *Advances in Computers*, Vol. 15 (Rubicoff and Yovits, eds.) Academic Press, New York, (1976), pp. 65–118.

[12] John R. Rice, and Ronald F. Boisvert, *Solving Elliptic Problems Using ELLPACK*, Springer-Verlag, New York, (1985).

[13] A. Samuel, Some studies in machine learning using the game of checkers, II: Recent progress, *IBM J. Research Devel.*, 11 (1967), pp. 601–617.

[14] Phillip D. Wasserman, *Neural Computing*, van Nostrand Reinhold, New York, 1989.