Machine Learning Techniques in Optimal Design

Giuseppe Cerbone
Oregon State University
Computer Science Dept.
Corvallis, OR 97331 (USA)

Introduction

Many important applications can be formalized as constrained optimization tasks. For example, we are studying the engineering domain of two-dimensional (2-D) structural design. In this task, the goal is to design a structure of minimum weight that bears a set of loads.

Figure 1 shows a solution to a design problem in which there is a single load (L) and two stationary support points (S1 and S2). The solution consists of four members, E1, E2, E3, and E4 that connect the load to the support points. In principle, optimal solutions to problems of this kind can be found by numerical optimization techniques. However, in practice [Vanderplaats, 1984] these methods are slow and they can produce different local solutions whose quality (ratio to the global optimum) varies with the choice of starting points. Hence, their applicability to real-world problems is severely restricted.

To overcome these limitations, we propose to augment numerical optimization by first performing a symbolic compilation stage to produce (a) objective functions that are faster to evaluate and that depend less on the choice of the starting point and (b) selection rules that associate problem instances to a set of recommended solutions. These goals are accomplished by successive specializations of the problem class and of the associated objective functions. In the end, this process reduces the problem to a collection of independent functions that are fast to evaluate, that can be differentiated symbolically, and that represent smaller regions of the overall search space. However, the specialization process can produce a large number of sub-problems. This is overcome by deriving inductively selection rules which associate problems to small sets of specialized independent sub-problems. Each set of candidate solutions is chosen to minimize a cost function which expresses the tradeoff between the quality of the solution that can be obtained from the sub-problem and the time it takes to produce it. The overall solution to the problem, is then obtained by solving in parallel each of the sub-problems in the set and computing the one with the minimum cost.

In addition to speeding up the optimization process, our use of learning methods also relieves the expert from the burden of identifying rules that exactly pinpoint optimal candidate sub-problems. In real engineering tasks it is usually too costly to the engineers to derive such rules. Therefore, this paper also contributes to a further step towards the solution of the knowledge acquisition bottleneck [Feigenbaum, 1977] which has somewhat impaired the construction of rule-based expert systems.

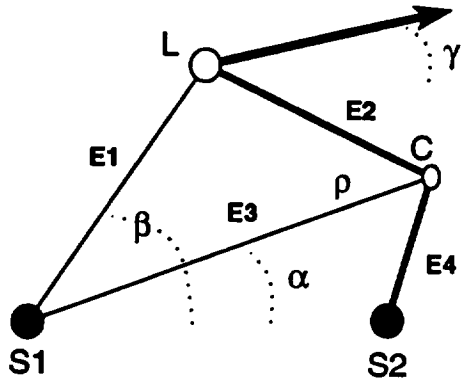

Figure 1: A solution to a 2-D structural design problem with given topology.

Our optimization schema differs from techniques currently used in the machine learning community. Our approach relies on the specialization of the problem via incorporation of constraints prior to optimization. Braudaway [Braudaway, 1988] designed a system along the same principle. However, to our knowledge, very little work has been done in using learning techniques to speedup numerical optimization tasks. In contrast, the current trend in the machine learning community focuses on methods, such as Explanation Based Learning (EBL) [Ellman, 1989], capable of generating rules. In addition, EBL methods have had little

success in the task of optimizing numerical procedures. We conjecture that one of the reasons is the dependence of EBL methods on the trace of the problem solver. The trace of a numerical optimizer gives little information on the structure of the problem. Therefore, in mathematical domains, EBL-derived rules are too detailed to produce any appreciable speedup.

The remainder of the paper is organized as follows. Section presents the 2-D structural design task. This is followed in Section by an overview of numerical optimization methods, their limitations, and our solution which is illustrated using a simple example. The machine learning methods are outlined in Section . These methods are then applied in Section which illustrates the experiments. These show that, for a certain family of problems, the compilation stage produces a substantial improvement in the performance of the optimization methods. Benefits and limitations of our strategy are summarized in Section , which also outlines future work.

Task description

Table 1 describes the 2-dimensional structural design task that we are attacking. Figure 1 shows an example problem in which L is the load and S1 and S2 are two supports. The so-called "topology" is given as a graph structure containing four edges (the members) and four vertices (the load, the two supports, and an intermediate connection point C). The topology does not specify the lengths of the members or the location of C. The topology and the position shown in the figure

Table 1: The 2-D Design Task.

Given:	A 2-dimensional region R A set of stable points (supports) A set of external loads with application points within R
Find:	The number of members, connectivity, and positions of all intermediate connection points such that the structure has minimum weight and is stable with respect to all external loads.

give the minimum-weight solution. In this solution, 4 members are used and E1 and E3 are in tension (they are being "stretched"), while members E2 and E4 are in compression. Tension members will be referred to as "rods" and indicated by thin lines. Compression members will be referred to as "columns" and indicated by thick lines. The type of members used in the solution is an abstraction that we have used throughout our work. To indicate a configuration of tensile and compressive members that consititutes a solution, we have defined the *stress state*. The stress state is an array of m elements in which each element corresponds to a member. The value of each element in the array is $+1$ if the member is tensile and -1 if the member is compressive.

The weight of a truss can be decreased in at least two ways. First, the engineer can use lighter material. Second, the "shape" can be designed in such a way that, for instance, it uses less material and, hence, it is lighter. In this paper we do not consider the (admittedly) important advances in the science of material but, instead, we focus on the synthesis of shapes that reduce the weight of a truss with a chosen construction material.

The task shown in Table 1 is actually only one step in the larger problem of designing good structures. In general, structural design proceeds in three steps [Palmer and Sheppard, 1970; Vanderplaats, 1984]. First, the problem solver chooses the topology, which specifies the locations of the loads and supports and the connectivity of the members. Then, the second step is to determine the locations of the connection points (and hence the lengths, locations, internal forces, and cross-sectional areas of the members) so as to minimize the weight of the structure. This is usually accomplished by numerical non-linear optimization techniques. The third and final step in the process optimizes the shapes of the individual members. This can often be accomplished by linear programming.

In addition to focusing only on the first two steps, we have introduced several simplifying assumptions to provide a tractable testbed for developing and testing machine learning methods. Specifically, we assume that structural members are joined by frictionless pins, only statically determinate structures are considered, the cross section of a column is square, columns and rods of any length and cross sectional area are available, and supports have no freedom of movement. A statically determinate structure contains no redundant members, and hence, the geometrical layout completely determines the forces acting in each member.

Given these assumptions, the weight of a candidate solution is usually calculated by a three-step process. The first step is to apply the *method of joints* [Wang and Salmon, 1984] to determine the forces operating in each member. Once this is known, the second step is to classify each member as compressive or tensile. This is important, because compressive and tensile members are composed of different materials and have different densities; e.g. concrete columns and high tensile steel rods. The third step is to determine the cross-sectional area of each member. The load that a member can bear is assumed to be linearly proportional to its cross-sectional area. Finally, the weight of each member can be computed as the product of the density of the appropriate material, the length of the member, and the cross-sectional area of the member.

The last two steps can be collapsed into a single parameter k: the ratio of the density per-unit-of-force-borne for compressive members to density per-unit-of-force-borne for tensile members. With this simplifica-

tion, instead of minimizing the actual weight, we can minimize the following quantity which, with an abuse of notation, we define as

$$Weight = \sum_{\substack{\text{tensile} \\ \text{members}}} \|F_i\| \, l_i + \sum_{\substack{\text{compressive} \\ \text{members}}} k \, \|F_j\| \, l_j .$$

F_i is the force in member i, and l_i is the length of member i. This is the initial objective function for the work described in this paper.

We conclude this section with a brief description of the method of joints, which is one of the methods used to calculate the F_i in statically determinate structures. The method of joints computes these forces by solving a system of linear equations as illustrated, for the problem in Figure 1, in Table 2. The matrix of coefficients is called [Wang and Salmon, 1984] the *axial* (or *static*) matrix and the vector of givens is defined as the *load vector*. In Figure 1, let C $= (x, y)$, S1 $= (x_1, y_1)$, and S2 $= (x_2, y_2)$, be the cartesian coordinates of the connection point, and the two supports, respectively. In addition, let (x_l, y_l) be the coordinates, and let p and γ be the magnitude and direction of the load L. The internal forces in each member are obtained by first constructing the axial matrix and load vector and then solving the system of equations for the unknown internal forces. Table 2 shows the symbolic system of equations for the example in Figure 1 with unknown forces F_1, F_2, F_3, and F_4 and with the coordinates of all the points explicitly substituted.

Now that we have defined the 2-dimensional design task and formulated it as a non-linear optimization problem, let us turn, in the next section, to a brief review of existing techniques for optimization and to the proposed methods.

Knowledge-based Optimization

Classical optimization textbooks [Vanderplaats, 1984; Papalambros and Wilde, 1988] present a comprehensive survey of optimization methods and of various techniques for conducting the search for an optimal solution. The schema illustrated in Figure 2 is typical of many domain independent non-linear optimization methods. The process is iterative. Starting at some initial point, the objective function is evaluated and the termination criteria are tested. If the test fails, a new point is generated by taking a step, of some chosen length in some chosen direction, away from the current point. Each point defines a set of values for the independent variables in the objective function.

Most optimization algorithms differ primarily in the criteria used to choose the direction along which to optimize. Some optimization methods (e.g., Powell's method [Vanderplaats, 1984]) choose the direction and step size using only evaluations of the objective function. Other methods, such as gradient descent and its variations [Papalambros and Wilde, 1988], require computation of the partial derivatives of the objective

Table 2: Method of Joints for the example in Figure 1. The product of the *axial matrix* and of the unknown forces F_i equals the *load vector*.

$$\begin{pmatrix} cos(\alpha_1) & cos(\alpha_2) & 0 & 0 \\ sin(\alpha_1) & sin(\alpha_2) & 0 & 0 \\ 0 & cos(\alpha_2 + 180) & cos(\alpha_3) & cos(\alpha_4) \\ 0 & sin(\alpha_2 + 180) & sin(\alpha_3) & sin(\alpha_4) \end{pmatrix} \times \begin{pmatrix} F_1 \\ F_2 \\ F_3 \\ F_4 \end{pmatrix} = \begin{pmatrix} Lcos(\gamma) \\ Lsin(\gamma) \\ 0 \\ 0 \end{pmatrix}$$

$$\begin{array}{llll} cos(\alpha_1) = (x_1 - x_l)/l_1, & cos(\alpha_2) = (x - x_l)/l_2 \\ cos(\alpha_3) = (x_1 - x)/l_3, & cos(\alpha_4) = (x_2 - x)/l_4 \\ sin(\alpha_1) = (y_1 - y_l)/l_1, & sin(\alpha_2) = (y - y_l)/l_2 \\ sin(\alpha_3) = (y_1 - y)/l_3, & sin(\alpha_4) = (y_2 - y)/l_4 \end{array}$$

and l_i's are Euclidean distances:

$$\begin{aligned} l_1 &= \sqrt{(x_1 - x_l)^2 + (y_1 - y_l)^2} \\ l_2 &= \sqrt{(x - x_l)^2 + (y - y_l)^2} \\ l_3 &= \sqrt{(x - x_1)^2 + (y - y_1)^2} \\ l_4 &= \sqrt{(x - x_2)^2 + (y - y_2)^2}. \end{aligned}$$

function to choose the new direction of optimization. Still other methods approximate the partial derivatives numerically by evaluating the objective function at many points.

The primary computational expense of numerical optimization methods is the repeated evaluation of the objective function. An advantage of gradient descent methods is that they need to evaluate the objective function less often, because they are able to take larger, and more effective steps. Of course, they incur the additional cost of repeatedly evaluating the partial derivatives of the objective function. Hence, they produce substantial savings only when the reduction in the number of function evaluations offsets the cost of evaluating the derivatives.

In engineering design, the objective function is typically very expensive to evaluate. This slows the numerical optimization process because the speed of numerical optimization is determined by the cost and frequency of evaluating the objective function. For the structural design domain to compute the objective function (volume of each structure) a system of linear equations must be solved. This is typically carried out by algorithms which are cubic in the number of unknowns. This number is usually large in real applications like bridge design. Furthermore, the fact that the constant k is applied only to compressive members makes it impossible to obtain a differentiable closed-form. The signs of the internal forces must be com-

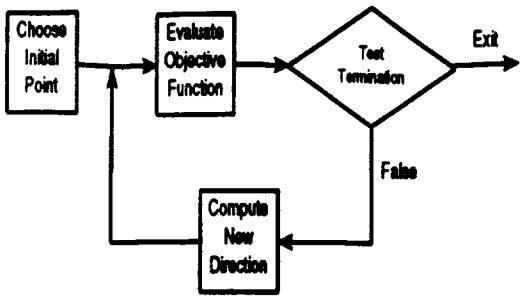

Figure 2: Traditional optimization schema.

puted before it is possible to determine which members are compressive. This prevents the use of gradient-based optimization methods that require fewer evaluations of the objective function – only slower function-based methods are applicable. One measure of the performance of a numerical optimizer is the time it takes to produce a solution. This quantity, however, depends on the choice of the starting point. Therefore, to obtain an accurate measurement, it is necessary to average the values obtained running the optimizer from different starting points.

Moreover, most engineering models are not unimodal. This directly affects the reliability of the solutions because numerical optimizers settle for local minima since they are unable to leap from one region to another to determine the global minimum. As shown in Figure 3, the objective function for the structural design domain is non unimodal. For instance, for the function in Figure 1 gradient methods started with $x = 1500$ and $y = 2000$ reach a local minimum in region R2 while the global minumum is in region R1. A measurement of the reliability can be obtained by taking the ratio (quality) of the local minimum and of the global minimum in controlled experiments in which the absolute minimum can be easily computed. Time and quality induce a tradeoff that can be exploited by defining the function:

$$\texttt{utility(solution)} = \texttt{CPUtime(solution)} * \texttt{CPUcost} + \texttt{quality(solution)}$$

where CPUcost is a positive constant that accounts for the cost of running the optimizer. We have used this definition in the learning stages of our approach to focus the attention of the optimization process on a few candidates that will produce solutions of maximum utility.

As shown in Figure 4, the increased reliability and speed are accomplished by augmenting the traditional run time optimization with a "compilation" stage prior to numerical optimization. The inputs to the compiler are (a) an high level description of the problem, (b)

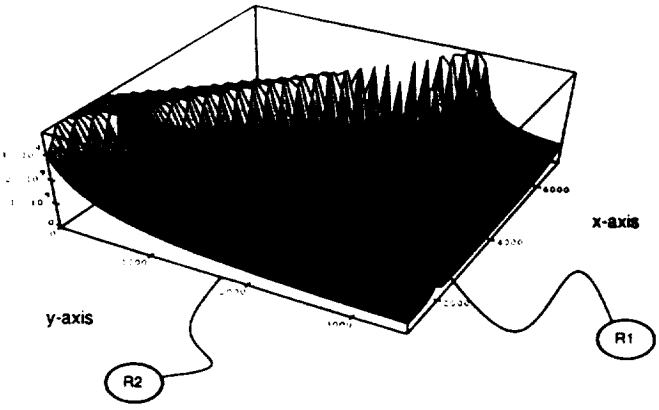

Figure 3: Volume of the structure in Figure 1.

domain knowledge about stress states, and (c) a procedure to generate training examples. Symbolic and inductive techniques are then used to (1) produce simplified versions of the objective function per each stress state, and (2) learn stress state selection rules which map problem instances into sets of candidate stress states of minimum cost.

First, the compiler produces one objective function for each topology and stress state. Each of these functions is a specialized version of the expression of the weight and it is faster to evaluate than the original, less specific, objective function. As an example, the function produced for the topology and stress state in Figure 1 is illustrated in Table 3. This expression is a closed form of the weight of a structure as a function of the two cartesian coordinates of connection point C restricted to region R1 in Figure 3. Moreover, these simplified expressions are differentiable and this permits the use of faster gradient-based optimization algorithms.

Another obstacle to practical applications of numerical optimization methods is the high dimensionality (number of independent variables) of the problems. Our compilation strategy decreases the dimensionality of optimization problems by searching a set of training examples for relations (regularities) among independent variables. These relations are then used as constraints among variables and are incorporated into the specialized versions of the objective function. This procedure eliminates independent variables with the result of greatly simplifying the optimization process, of enlarging its scope of applicability, and of speed-

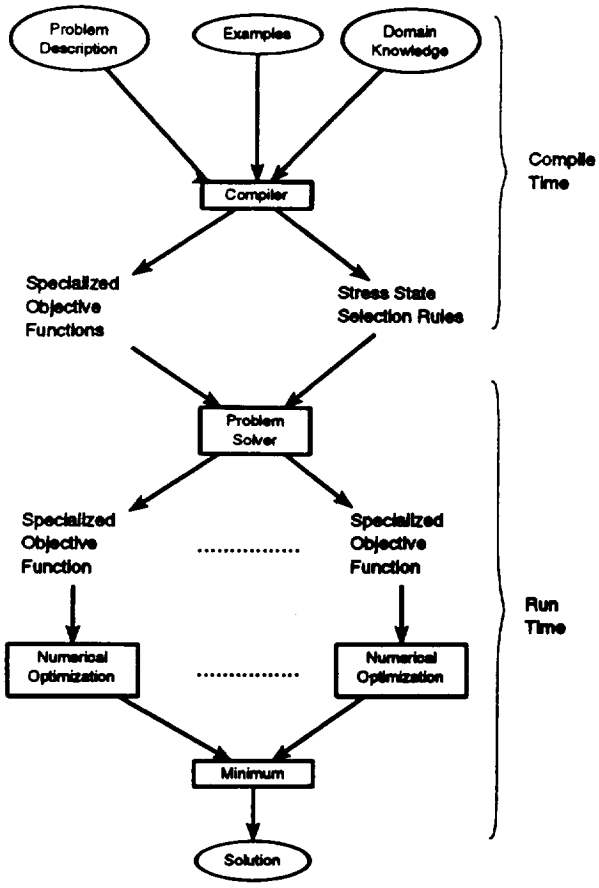

Figure 4: Proposed numerical optimization framework.

ing up run time optimization. For the region R1 in Figure 3, the compiler will determine that if the connection is expressed in polar coordinates ρ and α only the distance ρ from support S1 need be determined (see Figure 1.) This is because, in the analysis of the examples, it will discover that the angle α can be computed as one half of the angle β which is one of the givens of the problem. The final objective function is shown in Table 4 which contains only a single variable ρ vs. the two (x and y) in the expression in Table 3. This final expression indicates a reduction in dimensionality because, at run time, the numerical optimizer will only need to determine the value of ρ to compute the position of the connection point.

Finally, the compiler learns search control knowledge in the form of IF-THEN-ELSE rules. This is then used at run time to select stress states that lead quickly to quasi-optimal

Table 3: Partially evaluated objective function for the problem of Figure 1.

$$
\begin{aligned}
&Weight = \\
&\left(1.14\ 10^{13}x - 5.66\ 10^{9}x^2 + 8.16\ 10^{5}x^3 + \right. \\
&\ 3.28\ 10^{13}y - 3.26\ 10^{9}xy + 2.44\ 10^{5}x^2y - \\
&\ 6.70\ 10^{9}y^2 + 8.16\ 10^{5}xy^2 + 2.44\ 10^{5}y^3 - \\
&\left.\ 4.08\ 10^{16}\right) / \\
&\left(1.28\ 10^{1}xy - 2.56\ 10^{4}x + 2.56\ 10^{4}y - \right. \\
&\left.\ 6.40\ y^2 - 2.56\ 10^{7}\right)
\end{aligned}
$$

solutions. The set of stress states is chosen so that the *utility* of the stress states is maximized. The utility is a function that combines the time it takes to produce a solution with its expected quality (ratio to the global minimum.) This function introduces a trade-off between quality and time that is exploited by the learning algorithm [Cerbone and Dietterich, 1992]. As an example, for the design problem in Figure 1 whose objective function is shown in Figure 3, the compiler derives search control knowledge that allows the problem solver to focus the attention of the numerical optimizer on regions R1 and R2 when the load is directed toward support S2 and away from support S1.

Machine Learning Methods

This section describes in greater detail the symbolic and inductive learning techniques. Inductive learning techniques are used to (a) simplify the optimization process by reducing the number of independent variables and (b) derive the stress state selection rules. The inductive methods rely upon knowledge about the partitioning of the design space and upon a set of training examples that, for many engineering tasks, can be generated by the compiler. A complete discussion of the compilaton stages can be found in [Cerbone, 1992].

Symbolic Methods. Symbolic techniques are used to incorporate into the objective function knowledge about stress states and knowledge discovered during inductive analysis. The goal is to produce an highly simplified and specialized objective function. This is accomplished by partial evaluation [Futamura, 1971], and loop unrolling [Burstall and Darlington, 1977] - two techniques widely used in high-end optimizing compilers. Partial evaluation incorporates constant values for variables into functions (or programs) and simplifies them. Loop unrolling unfolds iterative con-

Table 4: Objective function for the structure in Figure 1 with reduced dimensionality.

$$
\begin{aligned}
&Weight_{simplified} \approx \\
&\left(1.16\ 10^{13}\rho - 5.19\ 10^{9}\rho^2 + 8.19\ 10^{5}\rho^3 - 4.08\ 10^{13}\right) / \\
&\ 3.95\rho^2
\end{aligned}
$$

structs (e.g., `for` loops) and transforms them into sequential programs. These techniques have been implemented using the Mathematica programming language [Wolfram, 1988] and [Maeder, 1989] which is suitable to numerical problems.

As an example of specialization, we illustrate how domain knowledge is used to specialize the objective function. First the problem solver chooses the topology. This can be simply done by enumerating a few possible configurations. Once the topology is chosen, it can be incorporated into the objective function. This allows us to compute symbolically the axial matrix and the load vector (see Section). We then apply symbolic algorithms to solve and simplify the system of equations and to obtain a closed-form expression for the forces. In principle, an infinite number of topologies should be explored; however, Friedland [Friedland, 1971] experimentally demonstrated that only a few of them need be considered to achieve satisfactory solutions.

The second specialization step is to plug in the givens of the problem and partially evaluate the resulting mixed symbolic/numeric expression. For our examples, the givens of the problems are the loads and supports; however, one may wish to analyze a structure subject to different inputs such as various loading conditions or support locations. In such cases it is possible to leave those values in symbolic form and substitute their numerical values at run time.

The third compilation step is to split the objective function V into cases according to stress state. When the objective function is specialized according to stress state, the result is a collection of special-case objective functions $\{V_1, \ldots, V_n\}$. Because each V_j corresponds to one stress state, it is possible to tell, at compile time, which forces should be multiplied by k. Hence, each V_j is differentiable, and this enables us to employ gradient-based optimization techniques that, typically, are faster than methods based only on evaluating the objective function alone.

Reduction of independent variables. A further speedup and increase in reliability of the numerical optimizers is obtained using inductive methods to decrease the number of independent variables (*dimensionality*) in the numerical optimization problem. The compiler is given a series of examples and uses them to inductively determine which independent variables can be computed as functions of known quantities. For instance, in the design domain, when searching within a region it might turn out to be superfluous to search along all dimensions because there might exist a simple relationship between one of the coordinates and known quantities like the location of loads and supports. These relations are then used as constraints and are incorporated into the objective functions. The result is the reduction of the number of independent variables. This, in turn, produces an even simpler and faster optimization problem. For instance, the function shown in Table 3 has two independent variables while the corresponding inductively simplified version has only one independent variable and it is shown in Table 4. Hence, the final optimization problem entails a simple linear optimization while the original one has two dimensions.

The variables to be eliminated are determined using an EBL-like approach which employs:

- training examples
- a library of given geometry entities (points, angles, etc.)
- a geometrical domain theory
- known relationships among geometric entities
- *regularities* – a mixture of heuristics and statistical regression techniques.

Each unknown connection point is subject to a compile time heuristic search process that attempts to compute (reformulate) the location as a function of loads and supports.

To see how this works, let us consider again the example problem in Figure 1 which we shall refer to as the "bisector" example. In this example, the connection point `C` is the unknown and the givens are the load `L` and the supports `S1` and `S2`. Moreover, let us assume that a set of training examples has been either provided or derived by the system. The reformulation starts by identifying all geometric objects using the given domain theory. For the bisector example, the system identifies, among others, the following geometric objects:

`point(S1), point(S2), point(C), point(L),`
`angle(`β`, L, S1, S2), angle(`α`, C, S1, S2),`
`segment(SG1, S1, S2), ...`

Predicates such as `point` and `angle` are basic elements of the given geometric domain theory. This means that, given a set of cartesian coordinates, the system is capable of computing each predicate. During the computation of each predicate, the system tags it as *given* or *unknown*. A predicate is *given* if all the entities used to compute it are either givens of the problem (loads or supports) or can be expressed a combination of given predicates. Otherwise, the predicate is tagged as *unknown*. For the bisector example, `point(C)` and all predicates that involve it in their derivation (e.g. `angle(`α`, C, S1, S2)`) are unknowns, all others are givens.

With this knowledge, the system then tries to relate the unknown geometric entity `point(C)` to as many other entities as possible with the ultimate goal of expressing it only using given geometric entities. This is accomplished by using a blend of EBL and discovery techniques. In the EBL jargon, the geometric knowledge base is the *domain theory*, `point(C)` is the target *concept*, and the operationality criterion is the fact that a concept must be expressed in terms of known geometric objects. To visualize this reformulation step, let us refer to the derivation tree in Figure 5.

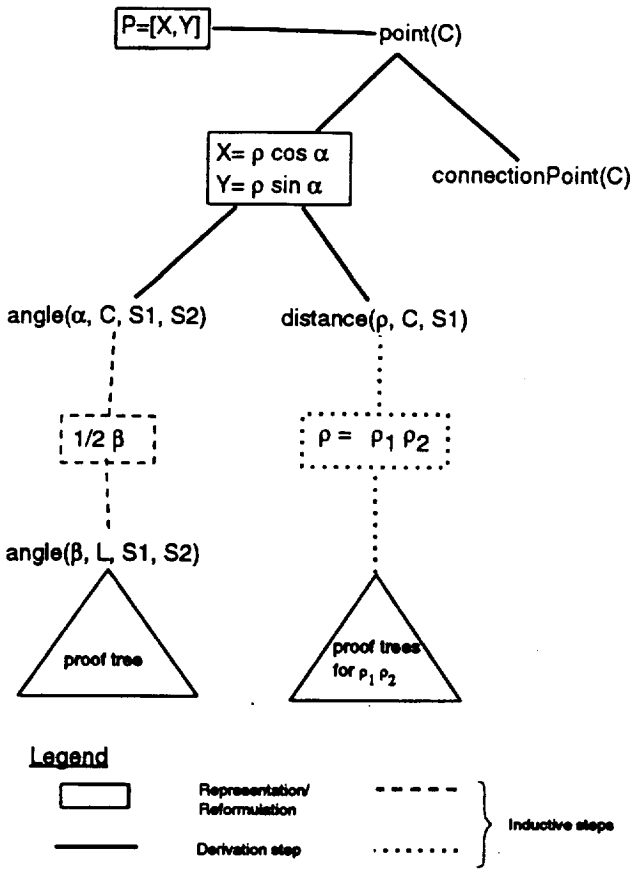

Figure 5: Decision tree to derive the *concept* `point(C)`.

The rightmost branch indicates that C is a connection point and, therefore, it is no longer explored. The leftmost branch, instead, uses a domain rule that reformulates a point in polar coordinates. Intuitively, the domain rule states that a point can be identified by its distance ρ from S1 and by the angle α between points C, S1, and S2. With this in mind, the system recursively tries to determine `angle(`α`, C, S1, S2)` and `distance(`ρ`, C, S1)`. After having exploited all proofs, the system concludes that it is not possible to re-express the `angle` and the `distance` in terms of known entities. If we were to follow EBL strictly, we should conclude that the domain theory is incomplete; that is, it is not powerful enough to bridge the gap between unknowns and givens. This, in turn, implies that the search would terminate concluding that `point(C)` cannot be re-expressed in terms of known geometric objects.

To overcome this problem we have used a discovery approach that fills these knowledge gaps with *eureka* [Burstall and Darlington, 1977] steps. Despite the name, however, in our strategy these steps are not arbitrary but inductive For the example in Figure 1, we determine that the angle α between points C, S1, and S2 is exactly one-half the angle β between points L, S1, and S2. Once this *regularity* is determined, in contrast with Burstall and Darlington's approach, we test the eureka step against all user provided examples to determine if it is a random occurance or a widespread phenomenon. In the former case, any use of this regularity is abandoned and others (if any) are tried. In the latter case, the regularity is assumed as a transformation of the unknown geometric entity. This is shown by the node in Figure 5 connected by the dashed lines. The system then subgoals on the geometric entities that were used to recognize the angle β. These are recognized as givens because they were derived from the position of the load and of the supports and the search terminates. The discussion of the branch identified by the dotted is similar to the one above and it is omitted for the sake of brevity.

The actual domain rules used in the geometric theory carry along also information that bridge the gap between the cartesian representation of a point and the polar one. This implies that the x and y coordinates of C can be expressed in terms of the angle α and of the distance ρ. In turn, the angle α is substituted by $\frac{\beta}{2}$ which can be computed from the given position of the load and supports. These transformations are considered as constraints and are incorporated into the objective function which is further simplified using the symbolic techniques. The result of the incorporation is shown in Table 4.

Rule derivation. The specialization steps discussed above greatly improve the running time of the optimizers on each objective function but they might introduce a large number of candidate solutions. These, in principle, can be exponential. To overcome this problem, we have devised a new inductive learning method to prune candidates that do not lead to optimal solutions. This method learns search control knowledge in the form of decision trees which can then be quickly transformed into `IF-THEN-ELSE` rules. These design rules associate features of the problem to a few regions in which the global minimum is believed to lie according to the examples given to the learning algorithms. The global solution is then obtained by running the optimizer on each of these regions and by taking the minimum solution.

We have found that most existing learning algorithms are not suitable for learning rules for optimization problems. The main obstacle is the absence of features that allow discrimination among classes. Algorithms like ID3 implicitely require independence of classes. Features with such discriminatory power are difficult to derive for many real application and especially for optimization tasks. On the other hand, it is relatively easy to provide *shallow* features which can circumscribe a set of possible solutions. Therefore, in devising our learning method we have assumed that all

features are *shallow* and proposed UTILITYID3, a novel learning algorithms. The algorithm resembles the well-known ID3 algorithm [Quinlan, 1987] in that it builds a decision trees and uses an information-theoretic heuristic to choose the feature on which to split at each recursive call. However, it is new in that the heuristic takes into consideration that the output is a set of recommended actions rather than a single discriminating class. This algorithm is fully described in [Cerbone, 1992] and [Cerbone and Dietterich, 1992].

In addition to the learning algorithm, we have introduced *maximum utility learning set*, a new learning framework. In this framework, a utility is associated to each candidate solution. The problem is to learn a set of actions of maximum utility that covers all given examples. For instance, in the design problem, the utility is a function of the time it takes the numerical optimizer to find a solution. The quality is measured with respect to the globally optimal design. It turns out that this learning problem is $\mathcal{NP}$ – *complete* [Garey and Johnson, 1979]. Hence, UTILITYID3 uses an approximation algorithm to determine a solution.

Experiments

To test the efficacy of this approach, we [Cerbone and Dietterich, 1991] have solved a series of design problems using an implementation based on Mathematica [Wolfram, 1988], and we have measured the impact of the compilation stages on the evaluation of the objective function, on the optimization task, and on the reliability of the optimization method. The measurements presented are averages over five randomly generated designs and, for each design, over 25 randomly generated starting points.

Objective function. The objective function of each design problem was evaluated in four different ways and, for each of them, we averaged the CPU[1] time over the different designs and starting points. The volume was first computed using the traditional, naive, numerical procedure with the method of joints. We then compiled the designs incorporating, in three successive stages, topological information, the givens of the problems, and the stress state. Figure 6 shows the time (per 100 runs) to evaluate the objective function at the various compilation stages. The biggest speedup was obtained with the numerical substitution of values into the symbolic closed form expression obtained and with the specialization to stress states. This suggests that the gain is related to the elimination of arithmetic operations from the original numerical problem.

Optimization. As indicated in Section , the running time of the optimizers is influenced by the number of function calls and by the time for each function evaluation. To present the benefits of our approach on the optimization task, we have experimented with two optimization algorithms (a) an optimizer based on Powell's method [Pike, 1986] that does not require gradient information and (b) the version of conjugate gradient descent [Press and others, 1988] provided by Mathematica. The graphs in Figures 7 and 8 report, respectively, the number of objective-function calls and the overall CPU time for each optimizer. The values connected by solid lines correspond to cases where the optimizer had no gradient information, while the values connected by dashed lines indicate averages utilizing the conjugate gradient descent method with alternative approximations for the gradient vector.

As expected, the number of evaluations remains constant throughout the compilation stages when the non-gradient is used, while it decreases drastically when we switch to the gradient-based optimization method.

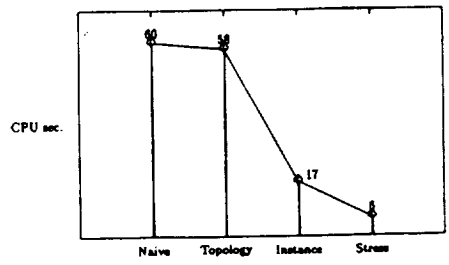

Figure 6: Influence of the compilation stage on the CPU time per function evaluation.

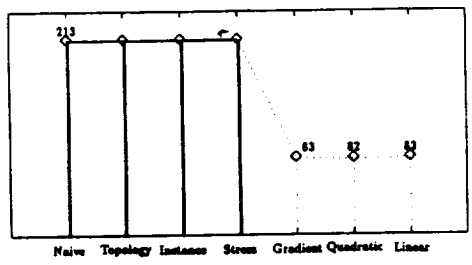

Figure 7: Influence of the compilation stage on the number of function calls.

The overall CPU time (Figure 8) steadily decreases as well. For the non-gradient method, the decrease is due to the progressive simplification of the objective

[1] The examples were run on a NeXT Cube with a 68030 board.

function itself, so that it is cheaper to evaluate. When we switch to the gradient method, there is initially no speedup at all, because the cost of evaluating the full gradient offsets the decrease in the number of times the objective function must be evaluated. However, additional speedups are obtained by approximating the objective function as a quadratic and as a linear function (by truncating its Taylor series).

We have found experimentally that there is no appreciable difference between the minima reached using the full gradient vector and the minima computed using quadratic approximations of the partial derivatives. However, the precision of the results obtained with the linear approximation is significantly reduced. Depending on the application, this trade of accuracy for speed may be acceptable. If not, the quadratic approximation should be employed.

Another possibility is to employ the linear approximation for the first half of the optimization search, and then switch to the quadratic approximation once the minimum is approached. In other words, the linear approximation can be applied to find a good starting point for performing a more exact search.

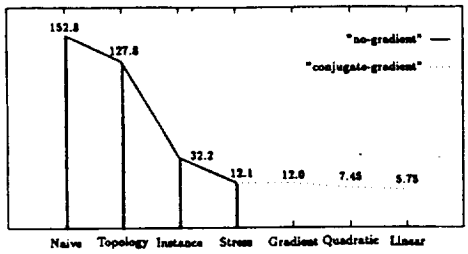

Figure 8: Influence of the compilation stage on the CPU time.

Reliability. An optimization method is reliable if it always finds the global minimum regardless of the starting point of the search. Unfortunately, as shown in Figure 3, the objective function in this task is not unimodal, which means that simple gradient-descent methods will be unreliable unless they are started in the right "basin." It is the user's responsibility to provide such a starting point, and this makes numerical optimization methods difficult to use in practice.

From inspecting graphs like Figure 3, it appears that, over each region corresponding to a single stress state, the objective function is unimodal. We conjecture that this is true for most of 2-D structural design problems. This means that optimization can be started from *any* point within a stress state, and it will always find the same minimum. If this is true, then our "divide-and-conquer" approach of searching each stress state in parallel will be guaranteed to produce the global optimum.

We have tested these hypothesis by performing 20 trials of the following procedure. First, a random starting location was chosen from one of the basins of the objective function that did not contain the global minimum. Next, two optimization methods were applied: the non-gradient method and the conjugate gradient method. Finally, our divide-and-conquer method was applied using, for each of the specialized objective functions V_j, a random starting location that exhibited the corresponding stress state. In all cases, our method found the global minimum while the other two methods converged to some other, local minimum.

Concluding Remarks

In this paper we have illustrated how machine learning techniques can be applied to optimal engineering design. This has been accomplished by tackling problems in two different areas:

- speeding up existing numerical methods
- learning a set of candidate optimal solutions.

Table 5 illustrates the correspondence between these problems and the machine learning techniques used in their solution. Our main contribution is to have shown that ML techniques can be effectively used to overcome some of the drawbacks of numerical optimizers and to increase their efficiency. Another contribution of this paper is to have shown that inductive techniques can complement traditional software engineering approaches in mathematical domains. This greatly reduces the need for knowledge transfer from experts to computer systems. In our approach, these results required the use of a blend of novel and traditional optimization techniques. First, we have defined a new learning framework which is more appropriate to optimization tasks. This framework involves (a) the requirement that the output of the learning algorithm be a set of alternatives and (b) measures of the cost of obtaining solutions. The learning methods produce sets of minimum cost. Within this framework we have developed algorithms which output IF-THEN-ELSE rules that associate problem characteristics (features) to sets

Table 5: Rows enumerate problems in optimal design. Columns list Machine Learning paradigms. X's indicate the ML paradigm used to solve the problem.

	Symbolic Methods	*Inductive Learning*
Selection Rules		X
Speedup of Numerical Optimizers	X	X

of optimal solutions. This is a contribution to basic research in machine learning. Second, we have demonstrated that inductive methods can also be used to simplify numerical problems. In fact we employed a discovery approach to reduce the number of independent variables. Finally, we have used more traditional compiler optimization techniques in a learning framework and merged them with inductive methods. We have shown that the overall result is a drastic speedup of the numerical optimization techniques.

Our approach opens new research directions into the so far unexplored area of applications of machine learning to numerical optimization. It is our hope that, in the medium-to long-term, our techniques will allow the use of specialized numerical optimizers in real-time applications like intelligent CAD systems.

Acknowledgements

The author wishes to thank his advisor, Thomas G.Dietterich, for the discussions that lead to this paper, David G. Ullman and Prasad Tadepalli for comments on related papers, Igor Rivin for information on the internals of Mathematica, and Jerry Keiper for insights into `FindMinimum`[] in Mathematica version 1.2. Ullman is responsible for suggesting the term *stress state*. This research was supported by NASA Ames Research Center under Grant Number NAG 2-630.

References

Braudaway, Wesley 1988. Constraint incorporation using constrained reformulation. Tech.Rep. LCSR-TR-100 Computer Science Dept., Rutgers University.

Burstall, R.M. and Darlington, J. 1977. A transformation system for developing recursive programs. *Journal of the ACM* 24(1):44–67.

Cerbone, Giuseppe and Dietterich, Thomas G. 1991. Knowledge compilation to speed up numerical optimization. In *Proceedings of the Machine Learning Workshop*. 600–604.

Cerbone, Giuseppe and Dietterich, Thomas G. 1992. Inductive learning in engineering: A case study. In *Proceedings of the Adaptive and Learning Systems Conference of the IEEE Society for Optical Engineers.*

Cerbone, Giuseppe 1992. *Machine Learning in Engineering: Techniques to Speed up Numerical Optimization.* Ph.D. Dissertation, Oregon State University, Corvallis, OR.

Ellman, Thomas 1989. Explanation-based learning: A survey of programs and perspectives. *ACM Computing Surveys* 21(2):163–222.

Feigenbaum, E.A. 1977. The art of artificial intelligence 1: themes and case studies of knowledge engineering. Tech.Rep. STAN-CS-77-621, Stanford University, Dept. of Computer Science.

Friedland, L.R. 1971. *Geometric Structural Behavior.* Ph.D. Dissertation, Columbia University at New York, N.Y.

Futamura, Y. 1971. Partial evaluation of a computation process – an approach to a compiler-compiler. *Systems, Computers, and Controls* 2(5):45–50.

Garey, Michael J. and Johnson, David S. 1979. *Computers and Intractability, A Guide to $\mathcal{NP}$-completeness.* Freeman.

Maeder, Roman 1989. *Programming in Mathematica.* Redwood City, Calif. : Addison-Wesley, Advanced Book Program.

Palmer, A.C. and Sheppard, D.J. 1970. Optimizing the shape of pin-jointed structures. In *Proc. of the Institution of Civil Engineers.* 363–376.

Papalambros, Panos Y. and Wilde, Douglass J. 1988. *Principles of optimal design: modeling and computation.* Cambridge University Press.

Pike, Ralph W. 1986. *Optimization for Engineering Systems.* Van Nostrand.

Press, William H. and others, 1988. *Numerical Recipes in C: the art of scientific computing.* Cambridge University Press, Cambridge.

Quinlan, Ross J. 1987. Simplifying decision trees. *International Journal of Man-Machine Studies* 27:221–234.

Vanderplaats, Garret N. 1984. *Numerical Optimization Techniques for engineering design with applications.* New York: McGraw Hill.

Wang, Chu-Kia and Salmon, Charles G. 1984. *Introductory Structural Analysis.* Prentice Hall, New Jersey.

Wolfram, Steven 1988. *Mathematica.* Wolfram Research.