Learning When Reformulation is Appropriate for Iterative Design

Mark Schwabacher Thomas Ellman Haym Hirsh Gerard Richter
Department of Computer Science
Hill Center for the Mathematical Sciences
Busch Campus
Rutgers, The State University of New Jersey
Piscataway, New Jersey 08855
{schwabac, ellman, hirsh, richter}@cs.rutgers.edu

Abstract

It is well known that search-space reformulation can improve the speed and reliability of numerical optimization in engineering design. We argue that the best choice of reformulation depends on the design goal, and present a technique for automatically constructing rules that map the design goal into a reformulation chosen from a space of possible reformulations. We tested our technique in the domain of racing-yacht-hull design, where each reformulation corresponds to incorporating constraints into the search space. We applied a standard inductive-learning algorithm, C4.5, to a set of training data describing which constraints are active in the optimal design for each goal encountered in a previous design session. We then used these rules to choose an appropriate reformulation for each of a set of test cases. Our experimental results show that using these reformulations improves both the speed and the reliability of design optimization, outperforming competing methods and approaching the best performance possible.

Keywords: mechanical engineering, design, decision tree induction, reformulation, numerical optimization.

1 Introduction

In a simulation-based automated engineering design system that uses numerical optimization, the decision on how to formulate the search space can dramatically affect the performance of the optimizer in two ways. First, using a lower-dimensional formulation of the search space makes optimization faster, since each gradient computation requires fewer runs of the simulator, and the distance in design space from the starting point to the optimum is smaller. In design problems where evaluating even just a single design can take tremendous amounts of time, selecting an appropriate formulation can be the determining factor in the success or failure of the design process. Second, different formulations of the search space can result in different degrees of "smoothness" of the

search space, which can impact not only the speed of the optimizer, but also the ability of the optimizer to get to the optimum, and therefore the quality of the resulting designs. We present a method of reformulation called "constraint incorporation," which reduces the dimensionality of the search space and increases its smoothness by incorporating constraints into the search space.

Traditionally, numerical optimization has dealt with explicit, "hard" constraints. The optimizer assumes that these constraints can never be violated. A hard constraint can be expressed as

$$f(x_1, x_2, \ldots, x_n) \leq k$$

(Here $x_1, x_2, \ldots, x_n$ are the *design parameters* that represent the design.) The constraint is said to be *inactive* if $f(x_1, x_2, \ldots, x_n) < k$, *active* if $f(x_1, x_2, \ldots, x_n) = k$, and *violated* if $f(x_1, x_2, \ldots, x_n) > k$. Hard constraints can result from the laws of physics, for example. Another type of constraint is the "soft" constraint, for which there is some sort of known penalty for violating the constraint. A soft constraint can be expressed as

if $f(x_1, x_2, \ldots, x_n) > k$ then apply penalty $P(x_1, x_2, \ldots, x_n)$

These usually arise from human-written laws, such as regulations specifying a monetary penalty for exceeding a certain noise level. In either case, if it is known that the constraint will be active at the optimal design point, and the constraint function f is invertible, then the constraint can be incorporated into the search space by using the inverse of f to eliminate one of the design parameters. [Papalambros and Wilde, 1988] describe how monotonicity knowledge can be used to determine that certain constraints will be active at the optimum. Incorporating these constraints produces a new search space with lower dimensionality, since the incorporation eliminates a design parameter, and greater smoothness, since the incorporation eliminates the "ridge" in the search space caused by the "if" statement in the constraint. If there are n constraints that can be incorporated in this way, then there are 2^n possible reformulations that can be produced by incorporating different subsets of constraints

Optimization can be done for a variety of *design goals*. A design goal consists of *environment parameters*, which are inputs to the simulator other than the design parameters, and the thresholds on the various constraints. Constraint activity depends on the goal (some constraints are active at the optimum for only some design goals), for two reasons. First, the constraint thresholds are part of the design goal. Second, different design goals will result in different optimal values of the design parameters on which the constraint functions depend. Because constraint activity depends on the goal, different reformulated search spaces are appropriate for different design goals. We describe a way in which inductive learning can be used to map the design goal into the appropriate reformulation.

2 Learning reformulation rules

The problem addressed by an inductive-learning system is to take a collection of labeled "training" data and produce rules that make accurate predictions on future data. To use inductive learning to form reformulation-selection rules, we take as training data a collection of design goals, each labeled with the set of constraints that are active at the optimal design point. We run the inductive

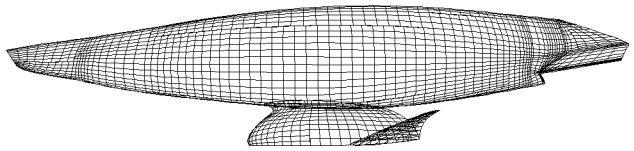

Figure 1: The Stars and Stripes '87.

learner once for each constraint, producing for each constraint a set of rules that can be used to predict whether the constraint will be active for new design goals.

Inductive learning is particularly suitable in the context of an automated design system because training data can be generated in an automated fashion. For example, one can choose a set of training goals and perform an optimization for each goal. One can then evaluate each constraint function for each optimal design, and then construct a table that records which constraints were active (within a threshold) for each training goal. This table can be used by the inductive-learning algorithm to generate a set of rules for each constraint, mapping the space of all possible goals into a prediction of whether or not that constraint will be active at the optimal design point for that goal. If learning is successful, these mappings extrapolate from the training data and can be used successfully in future design sessions to map a new goal into an appropriate reformulation.

The specific inductive-learning system used in this work is C4.5 [Quinlan, 1993] (release 6.0). The approach taken by C4.5 is to find a small decision tree that correctly classifies the training data, and to then remove lower portions of the tree that appear to fit noise in the data. The resulting tree is then used to assign labels to future, unlabeled data.

3 Yacht design

Our reformulation-selection techniques have been developed as part of the "Design Associate," a system for assisting human experts in the design of complex physical engineering structures [Ellman *et al.*, 1992]. One of the domains in which The Design Associate is currently being tested is the domain of 12-meter racing yachts, which until recently was the class of sailboats raced in America's Cup competitions. An example of a 12-meter yacht, the Stars and Stripes '87, is shown in Figure 1.[1]

Racing yachts can be designed to meet a variety of objectives, such as course time or cost. In our work we have chosen to focus on a course-time goal, namely minimizing the time it takes for a yacht to traverse a given race course under given wind conditions. A particular course-time goal thus requires the specification of two environment parameters: (1) the race course, represented as a set of $(\mathit{distance}, \mathit{heading})$ pairs; and (2) the wind speed, represented as a scalar number, in knots. Our design system represents a yacht geometry by a set of design parameters, and evaluates course time

[1] This is the boat that won the 1987 America's Cup competition, returning the trophy to the United States after an Australian win in 1983 (which represented the only non-US win in more than 100 years [Letcher *et al.*, 1987].)

using a "Velocity-Prediction Program" called "RUVPP,"[Schwabacher *et al.*, 1994] a somewhat simplified version of "AHVPP" from AeroHydro, Inc., which is a marketed product used in yacht design [Letcher, 1991].

Yacht designs are modified by operators that manipulate design parameters. A search space is thus specified by providing the parameters that define an initial prototype, and a set of operators for modifying that prototype.

To find a yacht for a given design goal our system uses CFSQP, a state-of-the-art implementation of the Sequential Quadratic Programming method [Craig *et al.*, 1994]. Sequential Quadratic Programming is a quasi-Newton method that solves a nonlinear constrained optimization problem by fitting a sequence of quadratic programs[2] to it, and then solving each of these problems using a quadratic programming method.

In the experiments described in this paper, we ran CFSQP with *course-time* as the objective function, and with one explicit, nonlinear, "hard" constraint. This constraint specifies that that the mass of the yacht, before adding any ballast, must be less than or equal to the mass of the water that it displaces. (In other words, the boat must not sink.)

Although the program we use to compute course time (RUVPP) is a state-of-the-art simulator, it nevertheless suffers from a number of deficiencies that make optimization difficult. For example, it will sometimes return a spurious root of the balance-of-force equations that it solves. It may also exhibit discontinuities, due to numerical round-off error, or due to truncation error in the numerical solver used to solve the balance-of-force equations. These deficiencies can produce "noise" in the evaluation function surface over which the optimization algorithm is moving. The algorithm can therefore easily get stuck at a point that appears to be a local optimum, but is nevertheless not locally optimal in terms of the true physics of the yacht design space. There is also noise in the search space caused by the constraints of the 12-Meter Rule, which is discussed further in the next section.

4 The reformulations

Yachts entered in the 1987 America's Cup race had to satisfy what is know as the 12-Meter Rule [IYRU, 1985]. The basic formula in the rule is:

$$\frac{length - freeboard + \sqrt{sailarea}}{2.37} \leq 12m$$

In addition to the basic formula, the rule contains several other constraints, along with associated penalties for violating these constraints. These constraints are:

- draft constraint
- beam constraint

[2] A quadratic program consists of a quadratic objective function to be optimized, and a set of linear constraints.

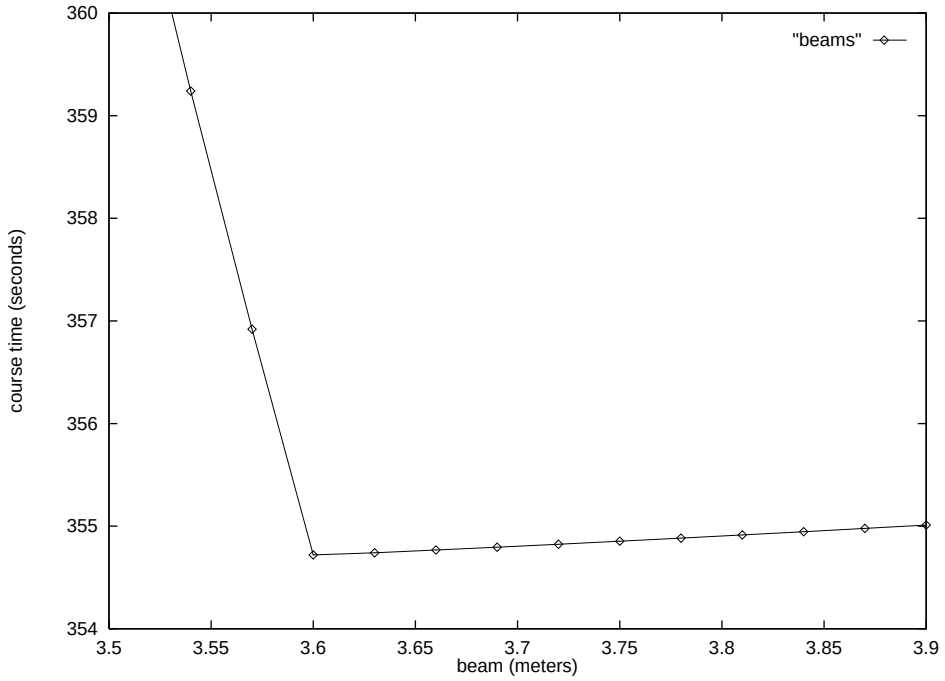

Figure 2: The nonsmoothness in the search space caused by the beam constraint.

- displacement constraint
- winglet span constraint

For example, the *beam constraint* states

> if $beam < 3.6m$, then add four times the difference to $length$

While constructing the simulator, we used a reasoning process similar to that described in [Papalambros and Wilde, 1988] to determine that the constraint described by the basic formula of the 12-Meter Rule, above, will always be active, since the objective function being minimized, *course-time*, is monotonically decreasing in *sail-area*, and the left-hand-side of the constraint is monotonically increasing in *sail-area*. We therefore *incorporated* this constraint into the simulator by solving for *sail-area* in terms of the other design parameters. So, for example, when the optimizer makes $length$ bigger, *sail-area* is automatically made smaller. In addition, because we also incorporated the other constraints into the simulator, reducing $beam$ beyond $3.6m$ causes the quantity *length* in the formula to increase, which causes *sail-area* to decrease.

Because the beam constraint contains an *if* statement, this incorporation causes a nonsmoothness in *course-time* as a function of $beam$. That is, there is a discontinuity in the first derivative of *course-time* with respect to $beam$. Figure 2 illustrates this nonsmoothness by showing the cross-section of the search space corresponding to the $beam$ design parameter. This nonsmoothness can cause a gradient-based optimizer such as CFSQP to get stuck, and to fail to get the optimum.

For many design goals, the optimal design is right on the constraint boundary. The optimal beam is often $3.6m$. If we expect the optimal beam to be $3.6m$, then we can incorporate the beam constraint into the operators. In the case of the beam constraint, this incorporation is trivial — we simply set $beam$ to $3.6m$ and leave it there. For other constraints, the incorporation is more complicated. For example, there is a constraint that specifies a penalty if $displacement$ does not vary with a certain cubic polynomial in $length$. $Displacement$ is not a design parameter; rather, it is a quantity computed from all of the design parameters. In order to incorporate the displacement constraint, we used Maple [Char *et al.*, 1992], a symbolic algebra package, to invert the displacement formula, and created a new set of operators that vary certain parameters while maintaining $displacement$ at the minimum displacement allowed by the constraint. For still-more-complicated constraints, it might not be possible to invert the constraint function using Maple; it might therefore be necessary for the operators to contain numerical solvers that find the right values of the incorporated design parameters to put the design on the constraint boundary.

We created operators to incorporate all four of the above-listed 12-Meter Rule constraints: the draft constraint, the beam constraint, the displacement constraint, and the winglet constraint. Using these operators, we are able to either incorporate or not incorporate each of these four constraints independently. We thus defined a set of sixteen possible reformulations of the search space. From our initial experiments with these operators, we determined empirically that incorporating the draft constraint substantially improved the reliability and speed of optimization for any design goal. We therefore decided to always incorporate the draft constraint, leaving us with a space of eight possible reformulations that we used in the experiments described below.

5 Learning to choose a reformulation

Having defined eight reformulations of the search space, we used inductive learning to decide, based on the design goal, which reformulation to use. As training data, we used 100 previous optimizations.[3] For each previous optimization, we evaluated each 12-Meter Rule constraint function at the optimum, and determined if the constraint was active (within a tolerance). Each of these previous optimizations had as its design goal minimizing course time for a single-leg racecourse, which can be represented using two numbers: the wind speed, and the heading (the angle between the yacht's direction and the wind direction). The design goal can therefore be represented using these two numbers. We ran the inductive learner once for each of the three constraints. Each time, the inductive learner was provided with a set of triples: the wind speed, the heading, and a ternary value indicating whether the constraint was inactive, active, or violated. One of the constraints was violated at the optimum in 10 of these optimizations. Figure 3 gives an example of a decision tree output by C4.5.

We used C4.5 to perform tenfold cross-validation [Weiss and Kulikowski, 1991], and obtained the error rates shown in Table 1. Here we compare the error rates of C4.5 with and without pruning, and of C4.5rules, a variant of C4.5 that extracts rules from the trees, with the expected error rate of random guessing (which is two-thirds since there are three classes from which to guess), and the

[3] The optimizer failed for one of these goals, so we used the remaining 99 goals as training data in the results that follow.

```
heading <= 109 :
|   windspeed <= 6.3 : active
|   windspeed > 6.3 :
|   |   windspeed > 8.2 : violated
|   |   windspeed <= 8.2 :
|   |   |   heading <= 65 : violated
|   |   |   heading > 65 : active
heading > 109 :
|   windspeed > 11.5 : active
|   windspeed <= 11.5 :
|   |   heading <= 135 : active
|   |   heading > 135 : inactive
```

Figure 3: Learned decision tree for the displacement constraint.

Table 1: Cross-validated error rates for selecting whether to incorporate each constraint.

Constraint	C4.5 w/ pruning	C4.5 w/o pruning	C4.5rules	MFC	Random
Beam	11.1%	11.1%	11.1%	33.3%	66.7%
Displacement	15.1%	15.1%	15.1%	53.5%	66.7%
Winglet	7.0%	10.0%	10.0%	13.1%	66.7%

error rate of the Most Frequent Class (MFC) learning method. MFC always chooses the class that occurs most frequently in the training data. In this case, that means that it always chooses the same reformulation, namely the one that is most often the best reformulation in the training data.

As Table 1 shows, C4.5 with pruning performed slightly better than C4.5 without pruning or C4.5rules (and so in our further experiments reported below we use only C4.5 with pruning), and all three significantly outperformed MFC, which in turn significantly outperformed random guessing.

However, these results are for error rates, the proportion of cases where learning makes an incorrect guess, and more important in this domain is how learning affects the overall problem-solving task, namely how it improves the speed and reliability of the design optimization process. Does learning make the design process faster or slower? Are the resulting designs better or worse? To measure these effects, we performed optimizations for 25 new randomly generated goals using the reformulations suggested by each learning method. Table 2 shows the effect that C4.5 (with pruning) and MFC had on the average course time (the quality of the design), and average number of evaluations (the speed of the optimization), as compared with the "old way" of doing optimization without incorporating any of the three constraints into the operators. We also include in this table the performance of several other methods. A hypothetical "omniscient" problem solver always magically guesses the best possible choice. No learning method will enable results superior to this.

Table 2: Effect of using reformulations chosen by learner on optimization performance.

method	quality change	time change
omniscient	+0.085%	-36%
exhaustive	+0.085%	+384%
C4.5	+0.080%	-35%
MFC	+0.029%	-32%
none	0	0
random	-0.276%	-40%
all	-0.599%	-74%

The "exhaustive" optimization method performs eight optimizations for each goal, using all eight possible reformulations, and then chooses the best resulting design. Incorporating "all" constraints all the time results in the fastest possible optimization within this set of reformulations (at the cost of quality loss).

C4.5 produced a significant speedup in optimization, with no quality loss. In fact, it produced a small quality increase. (This quality increase suggests that without any reformulation, the optimizer gets "stuck" on the "ridges" that the constraints cause the search space to have, and therefore sometimes fails to get the optimum.) MFC produced a slightly smaller speedup and a slightly smaller quality improvement. The difference between C4.5 and MFC was, however, statistically significant at the 99% confidence level, according to the paired t-test. Both learning methods performed substantially better than random guessing. C4.5 performed almost as well as the hypothetical omniscient learner, which means it performed almost as well as any learner could possibly do. Interestingly, according to the t-test, the difference between C4.5 and the omniscient method was not statistically significant, but this just illustrates a limitation of the t-test, since we know that the omniscient method really is better, on average, than C4.5.

Incorporating all of the constraints all of the time resulted in a very large speedup, with a modest quality loss. This method may be appropriate if one wants a quick and approximate optimization. It might, for example, be used in the early stages of design when the engineer wants to get a feel for the search space by asking "what-if" questions.

One question that these results raise is how training-data quantity affects performance. If one does not have results from a large number of previous optimizations available, then one can either run some extra optimizations to generate training data (which is expensive), or do the learning with less training data (which is likely to produce higher error rates and lower optimization performance). To explore this issue we applied our learning approach to datasets of varying sizes, with the error rates shown in Figure 4. For each training-set size in the figure, we randomly chose 10 different subsets of our training data of that size, and performed 10-fold cross-validation on each subset. The figure shows the averages. C4.5 outperformed MFC for every training-set size, but C4.5's error rate

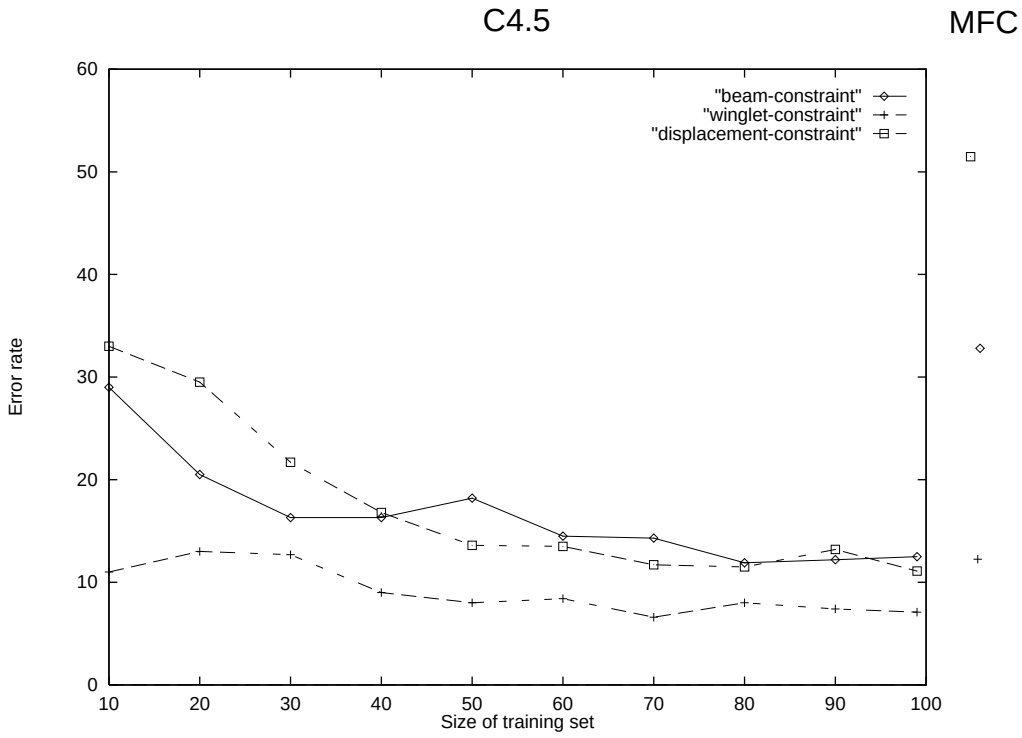

Figure 4: Effect of training set size on learner performance.

on smaller training sets was significantly larger than C4.5's error rate for larger training sets (with performance reaching an asymptote for training sets of about 60 cases or more).

6 Related work

Previously, we published results [Schwabacher *et al.*, 1994] showing that machine learning can improve optimization performance by learning how to select an initial prototype from which to start the optimizer. Cerbone [Cerbone, 1992] has reported work which applied machine-learning techniques to a problem similar to our prototype-selection problem. His design space, in the domain of truss design, has an exponential number of disconnected search spaces. He uses inductive learning techniques to learn rules for selecting a subset of these search spaces for further exploration. Several investigators [Orelup *et al.*, 1988, Tong, 1988, Powell, 1990, Hoeltzel and Chieng, 1987] have developed alternative artificial-intelligence techniques for controlling iterative parameter-design optimization. [Gelsey and Smith, 1995] describe a Search Space Toolkit which assists in determining properties of the search space that can be used for reformulation. [Choy and Agogino, 1986] describe a system that automates [Papalambros and Wilde, 1988]'s method of using monotonicity analysis to detect constraint activity. As far as we know, nobody has applied machine learning to predicting constraint activity, or to selection of a search space reformulation for design optimization.

7 Future work

This paper has described on-going work, and there are thus a number of directions for future work. These fall into two groups: extending this work to more difficult design tasks, and improving results by using other learning methods.

7.1 Other design tasks

The results presented here apply to a constrained class of yacht-design goals, those comprised of a single leg. One question is how this approach can be applied to courses comprised of varying numbers of legs. We believe that we could get reasonable optimization performance by using the trees learned from single-leg courses to perform multi-leg optimization in the following way: If a constraint should be incorporated for every leg of the racecourse, then incorporate it for the full, multi-leg course. We need to test how well optimization performs when handling racecourses in this manner. We could also attempt to learn directly for multi-leg racecourses. Doing so would raise an interesting machine-learning question, since describing a multi-leg racecourse requires a variable number of attributes, and thus traditional learners such as C4.5 do not directly apply.

We believe that the results presented here will easily generalize to situations in which there are more than eight reformulations. We used the results from the same set of 100 optimizations to perform three separate learning tasks (for three constraints), and then combined the rules generated by these three learning sessions to select one of the eight reformulations. As the number of reformulations grows, the number of constraints, and therefore the amount of CPU time needed for the learning, will grow logarithmically with the number of reformulations. The CPU time needed for learning is currently insignificant compared with the CPU time needed for the subsequent optimizations. We expect that as the number of reformulations grows, the number of training examples needed will remain constant (since the same training examples are used for each constraint), and the amount of CPU time needed for learning will remain insignificant. We plan to test this hypothesis by using other constraints within the yacht design domain, such as the "boat doesn't sink" constraint. We also plan to test our method for reformulations that do not involve explicit constraints. For example, the system might notice that an intermediate quantity in the simulator frequently has the same value, and learn when to constrain that intermediate quantity to have that value.

The learning approach could also be used to decide when to reformulate soft constraints as hard constraints. If it were known with a high degree of confidence that a certain soft constraint will not be violated at the optimum for certain goals, then this soft constraint could be converted into a hard constraint for those goals, which would eliminate a ridge from the search space and thereby make optimization more robust (although it would not reduce the dimensionality of the search space). For example, in the training data that we collected, the *beam constraint* was never violated, so it might be replaced safely with a hard constraint.

Other more-difficult problems might involve a less-smooth search space, a higher-dimensional goal space, or a less reliable optimizer. Such problems may arise when we test this method in other domains. In particular, we plan to test it in the domain of aircraft design. We also plan to explore how reformulated search spaces can be used in more complicated optimization strategies.

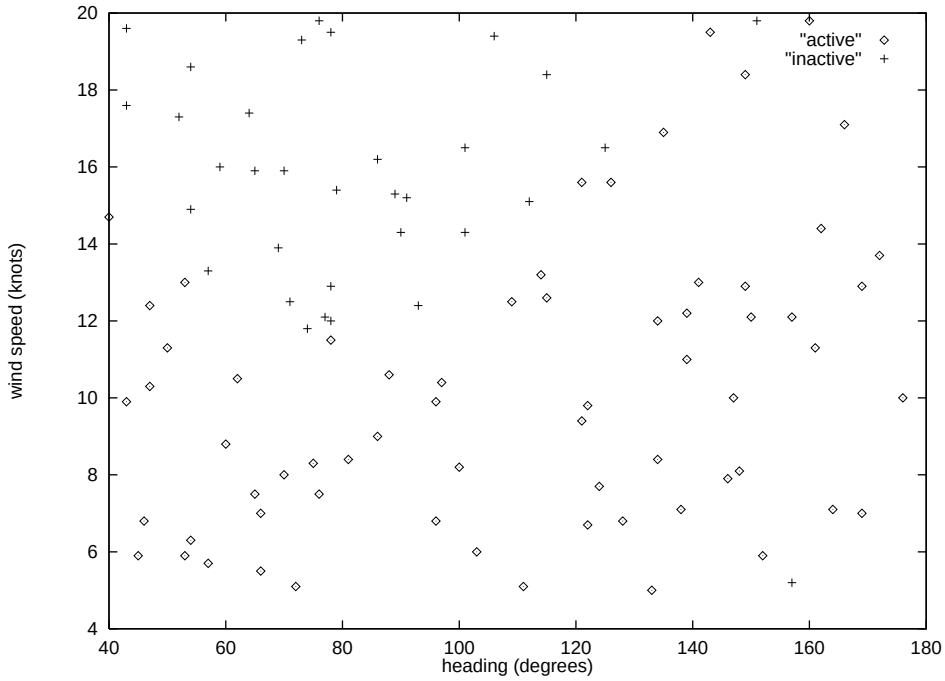

Figure 5: Activity of the beam constraint over the goal space.

For example, it might pay to first optimize in a lower-dimensional reformulated space, and then re-optimize in a higher-dimensional space, in the hope that the first optimization would quickly get close to the optimum of the higher-dimensional space.

7.2 Other learning methods

Our results are based on C4.5, but there are many other learning methods that could also be applied. For example, it would be interesting to see how well neural networks, nearest-neighbor methods, or statistical regression would perform. In particular, C4.5, like most decision-tree learners, uses linear, axis-parallel cuts in its decision trees. However, Figure 5 shows how the activity of the beam constraint varies over the goal space in the training data we used — the space is clearly divided into two regions (except for one point which we believe is noise). The border between these regions does not appear to be axis parallel, and appears to be nonlinear. This suggests that better performance might be achieved using an "oblique" decision tree learner, such as OC1 [Murthy *et al.*, 1993], or by attempting to learn nonlinear region boundaries.

As would be expected, even though our yacht-domain results with C4.5 were nearly optimal for 100 examples, results degrade when given less training data. Although it would be interesting to see if other learning methods would have better small-dataset performance, for any learner we would expect performance to be inferior for small enough datasets. One approach for improving results in such small-dataset cases — as well as in other cases where off-the-shelf learners such as C4.5 may not perform well even if given larger datasets — is to integrate background knowledge into the learning process. One form of background knowledge that is often available, such as in the yacht-design domain, is *modality constraints*. This is knowledge that expresses the modality of the

learned class with respect to the attributes. For example, we believe that optimal *beam* is monotonically increasing in wind speed, and monotonically decreasing in heading. We also know that the activity of any constraint of the form $f(x_1, x_2, \ldots, x_n) \leq k$ must be monotonic in k, so, for example, the activity of a cost constraint must be monotonic in the cost threshold. One open question is how such knowledge could be integrated into learning. One approach would be to use such modality constraints to remove from the training data points that violate the constraints (on the assumption that these points are noise). A second approach is to modify the tree induction algorithm so that it will never construct a tree that violates the constraints.

Finally, even after our learning approach is applied, every additional future optimization can serve as an additional training point for the learning. Thus learning methods that can work in an incremental fashion might also prove useful for this task. In addition, it may prove useful to develop methods that select suitable data prior to learning. For example, when there are not enough existing optimizations to achieve adequate learning results, additional optimizations can be performed to generate further training data. Rather than performing these new optimizations for random goals or for a set of goals that span the goal space, one could allow the learner to choose the goals to be used in the new training data. Background knowledge — such as modality constraints — could prove particularly useful in selecting such goals.

Acknowledgments

This research has benefited from numerous discussions with members of the Rutgers HPCD project. In particular, we would like to thank Andrew Gelsey, John Keane, and Brian Davison. This research is part of the Rutgers-based HPCD (Hypercomputing and Design) project supported by the Advanced Research Projects Agency of the Department of Defense through contract ARPA-DAST 63-93-C-0064.

References

[Cerbone, 1992] G. Cerbone. Machine learning in engineering: Techniques to speed up numerical optimization. Technical Report 92-30-09, Oregon State University Department of Computer Science, 1992. Ph.D. Thesis.

[Char *et al.*, 1992] B.W. Char, K.O. Geddes, G.H. Gonnet, B.L. Leong, M.B. Monagan, and S.M. Watt. *First Leaves: A Tutorial Introduction to Maple V*. Springer-Verlag and Waterloo Maple Publishing, 1992.

[Choy and Agogino, 1986] J. Choy and A. Agogino. Symon: Automated symbolic monotonicity analysis system for qualitative design optimization. In *Proceedings ASME International Computers in Engineering Conference*, 1986.

[Craig *et al.*, 1994] L. Craig, J. Zhou, and A. Tits. User's guide for cfsqp version 2.1: A c code for solving (large scale) constrained nonlinear (minimax) optimization problems, generating iterates satisfying all inequality constraints. Technical Report TR-94-16r1, Institute for Systems Research, University of Maryland, November 1994.

[Ellman *et al.*, 1992] T. Ellman, J. Keane, and M. Schwabacher. The rutgers cap project design associate. Technical Report CAP-TR-7, Department of Computer Science, Rutgers University, New Brunswick, NJ, 1992.

[Gelsey and Smith, 1995] A. Gelsey and D. Smith. A search space toolkit. In *Proceedings of the Eleventh Conference on Artificial Intelligence for Applications*, 1995.

[Hoeltzel and Chieng, 1987] D. Hoeltzel and W. Chieng. Statistical machine learning for the cognitive selection of nonlinear programming algorithms in engineering design optimization. In *Advances in Design Automation*, Boston, MA, 1987.

[IYRU, 1985] *The Rating Rule and Measurement Instructions of the International Twelve Metre Class*. International Yacht Racing Union, 1985.

[Letcher *et al.*, 1987] J. Letcher, J. Marshall, J. Oliver, and N. Salvesen. Stars and stripes. *Scientific American*, 257(2), August 1987.

[Letcher, 1991] J. Letcher. *The Aero/Hydro VPP Manual*. Aero/Hydro, Inc., Southwest Harbor, ME, 1991.

[Murthy *et al.*, 1993] S. Murthy, S. Kasif, S. Salzberg, and R. Beigel. Oc1: Randomized induction of oblique decision trees. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, Washington, D.C., 1993.

[Orelup *et al.*, 1988] M. F. Orelup, J. R. Dixon, P. R. Cohen, and M. K. Simmons. Dominic ii: Meta-level control in iterative redesign. In *Proceedings of the National Conference on Artificial Intelligence*, pages 25–30, St. Paul, MN, 1988.

[Papalambros and Wilde, 1988] P. Papalambros and J. Wilde. *Principles of Optimal Design*. Cambridge University Press, New York, NY, 1988.

[Powell, 1990] D. Powell. Inter-gen: A hybrid approach to engineering design optimization. Technical report, Rensselaer Polytechnic Institute Department of Computer Science, December 1990. Ph.D. Thesis.

[Quinlan, 1993] John Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, CA, 1993.

[Schwabacher *et al.*, 1994] M. Schwabacher, H. Hirsh, and T. Ellman. Learning prototype-selection rules for case-based iterative design. In *Proceedings of the Tenth IEEE Conference on Artificial Intelligence for Applications*, San Antonio, Texas, 1994.

[Tong, 1988] S. S. Tong. Coupling symbolic manipulation and numerical simulation for complex engineering designs. In *International Association of Mathematics and Computers in Simulation Conference on Expert Systems for Numerical Computing*, Purdue University, 1988.

[Weiss and Kulikowski, 1991] Sholom M. Weiss and Casimir A. Kulikowski. *Computer Systems That Learn*. Morgan Kaufmann, San Mateo, CA, 1991.