Learning Prototype-Selection Rules for Case-Based Iterative Design

Mark Schwabacher Haym Hirsh Thomas Ellman
Department of Computer Science
Rutgers University
New Brunswick, NJ 08903

Abstract

The first step for most case-based design systems is to select an initial prototype from a database of previous designs. The retrieved prototype is then modified to tailor it to the given goals. For any particular design goal the selection of a starting point for the design process can have a dramatic effect both on the quality of the eventual design and on the overall design time. We present a technique for automatically constructing effective prototype-selection rules. Our technique applies a standard inductive-learning algorithm, C4.5, to a set of training data describing which particular prototype would have been the best choice for each goal encountered in a previous design session. We have tested our technique in the domain of racing-yacht-hull design, comparing our inductively learned selection rules to several competing prototype-selection methods. Our results show that the inductive prototype-selection method leads to better final designs when the design process is guided by a noisy evaluation function, and that the inductively learned rules will often be more efficient than competing methods.

1: Introduction

Many automated design systems begin by retrieving an initial prototype from a library of previous designs, using the given design goal as an index to guide the retrieval process [14]. The retrieved prototype is then modified by a set of design modification operators to tailor the selected design to the given goals. In many cases the quality of competing designs can be assessed using domain-specific evaluation functions, and in such cases the design-modification process is often accomplished by an optimization method such as hill-climbing search [12, 2]. Such a design system can be seen as a *case-based reasoning* system [4], in which the prototype-selection method is the *indexing* process, and the optimization method is the *adaptation* process.

In the context of such case-based design systems, the choice of an initial prototype can affect both the quality of the final design and the computational cost of obtaining that design, for three reasons. First, prototype selection may impact quality when the prototypes lie in disjoint search spaces. In particular, if the system's design modification operators cannot convert any prototype into any other prototype, the choice of initial prototype will restrict the set of possible designs that can be obtained by *any* search process. A poor choice of initial prototype may therefore lead to a suboptimal final design. Second, prototype selection may impact quality when the design process is guided by a nonlinear evaluation function with unknown global properties. Since there is no known method that is guaranteed to find the global optimum of an arbitrary nonlinear function [7], most design systems rely on iterative local search methods whose results are sensitive to the initial starting point. Finally, the choice of prototype may have an impact on the time needed to carry out the design modification process—two different starting points may yield the same final design but take very different amounts of time to get there. In design problems where evaluating even just a single design can take tremendous amounts of time, selecting an appropriate initial prototype can be the determining factor in the success or failure of the design process.

This paper describes the application of inductive learning [11] to form rules for selecting appropriate prototype designs. The paper is structured as follows. In Section 2, we describe our inductive method for learning prototype-selection rules. In Section 3 we describe the domain of racing-yacht-hull design, in which we tested our prototype-selection methods. In Sections 4 and 5, we describe the experiments

This research has benefited from numerous discussions with members of the Rutgers CAP project. We thank Andrew Gelsey for helping with the cross-validation code, John Keane for helping with RUVPP, and Andrew Gelsey and Tim Weinrich for comments on a previous draft of this paper. This research was supported under ARPA-funded NASA grant NAG 2-645.

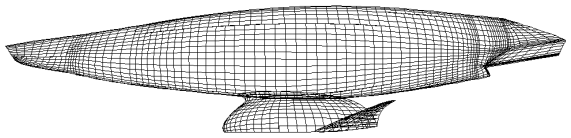

Figure 1: The Stars and Stripes '87.

that we performed to test our methods and compare them with four alternative prototype-selection methods. Section 6 contains a discussion of the computational complexity of our approach. Sections 7 and 8 review related work and present ideas for future work.

2: Learning prototype-selection rules

The problem addressed by an inductive-learning system is to take a collection of labeled "training" data and form rules that make accurate predictions on future data. To use inductive learning to form prototype-selection rules, we take as training data a collection of design goals, each labeled with which prototype in the library is best for that goal. "Best" can be defined to mean the prototype that best satisfies the design objectives, the prototype that results in the shortest design time, or the prototype that optimizes some combination of design quality and design time.

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 all combinations of training goals and library prototypes. One can then construct a table that records which prototype was best for each training goal. This table can be used by the inductive-learning algorithm to generate rules mapping the space of all possible goals into the set of prototypes in the library. If learning is successful this mapping extrapolates from the training data and can be used successfully in future design sessions to map a new goal into an appropriate initial prototype in the design library.

The specific inductive-learning system used in this work is C4.5 [8] (release 3.0, with windowing turned off). The approach taken by C4.5 is to find a small decision tree that correctly classifies the training data, then remove lower portions of the tree that appear to fit noise in the data. The resulting tree is then used as a decision procedure for assigning labels to future, unlabeled data.

3: Yacht design

Our prototype-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 [2]. The Design Associate is currently being tested in the domain of 12-meter racing yachts, which until recently was the class of yachts raced in America's Cup competitions. An example of a 12-meter yacht, the Stars and Stripes '87, is shown in Figure 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 things: (1) the race course, represented as a set of (*distance*, *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 B-spline surfaces [13], and evaluates course time using a "Velocity-Prediction Program" called "AHVPP" from AeroHydro, Inc., which is a marketed product used in yacht design [5].

Yacht designs are modified by operators that manipulate the B-spline surfaces. A search space is thus specified by providing an initial prototype geometry and a set of operators for modifying that prototype. Our current set of shape-modification operators was obtained by asking our yacht-design collaborators for an exhaustive list of all features of a yacht's shape that might be relevant to the racing performance of a yacht. These operators include

- Global-Scaling Operators: *Scale-X*, *Scale-Y* and *Scale-Z* change the overall dimensions of a racing yacht, by uniformly scaling all surfaces.
- Prismatic-Coefficient Operators: *Prism-X*, *Prism-Y* and *Prism-Z* make a yacht's canoe-body more or less streamlined, when viewed along the X, Y and Z axes respectively.
- Keel Operators: *Scale-Keel* and *Invert-Keel* change the depth and taper ratio of the keel respectively.

These eight operators represent a subset of the full set that were actually developed, focusing on a smaller set suitable for testing our prototype-selection methods.

To find a yacht for a given design goal our system uses steepest-descent hillclimbing [7]. The steepest-descent algorithm operates by repeatedly computing

the gradient of the evaluation function. The algorithm then takes a step in the direction of the gradient, and evaluates the resulting point. If the new point is better than the old one, the new point becomes the current one, and the algorithm repeats. The algorithm terminates when the gradient is zero.

A number of enhancements to the hillclimbing algorithm have been adopted to deal with some practical difficulties arising in the yacht design domain. Although the program we use to compute course time (AHVPP) is a commercial software product, it nevertheless suffers from a number of deficiencies that make hillclimbing 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 discretization of the (theoretically) continuous yacht hull surface. These deficiencies can produce "noise" in the evaluation function surface over which the hillclimbing 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. To overcome these difficulties, we have endowed the hillclimbing algorithm with the ability to climb over hills of limited height and limited width. The resulting algorithm is more robust than the original algorithm; however, it still does not always reach a true local optimum. The significance of this for prototype selection is discussed further in Section 5.

4: Results

To test our prototype-selection method we conducted several sets of experiments. In each case we compare our approach with each of four other methods:

Closest goal. This method requires a measure of the distance between two goals, and knowledge of the goal for which each prototype in the design library was originally optimized. It chooses the prototype whose original goal has minimum distance from the new goal. Intuitively, in our yacht-design problem this method chooses a yacht designed for a course and windspeed most similar to the new course and windspeed.

Best initial evaluation. This method requires running the evaluation function on each prototype in the database. It chooses the prototype that, according to the evaluation function, is best for the new goal (before any operators have been applied to the prototype). In the case of our yacht-design problem this corresponds to starting the design process from whichever yacht in the library is fastest for the new course and windspeed.

Most frequent class. This is actually a very simple inductive method that always chooses a fixed prototype, namely the one that is most frequently the best prototype for the training data.

Random. This method involves simply selecting a random element from the design library, using a uniform distribution over the designs.

We compare these methods using two different evaluation criteria:

Error rate. How often is the wrong prototype selected?

Course-time increase. How much worse is the resulting average course time than the optimal choice that an omniscient selection would make?

In our experiments we focus primarily on the question of how well our inductive-learning prototype-selection method handles problems where the prototypes lie in disjoint search spaces. Our experiments therefore explore how prototype selection affects the quality of the final design.

For our first set of experiments we created a database of four designs that would serve as our sample prototype library (and thus also serve as the class labels for the training data given to our inductive learner). To achieve the goal of having each prototype define a different space, the design library was created by starting from a single prototype (the Stars and Stripes '87) and optimizing for four different goals using all eight of the operators given earlier. All subsequent design episodes used only four of the eight operators, so that each yacht would define a separate space.[1]

To test the learned prototype-selection rules we defined a space of goals consisting of a windspeed and a racecourse, where the windspeed is constrained to be 8, 10, 12, 14, or 16 knots, the racecourse is constrained to be 80% in one direction, and 20% in a second direction, and each direction is constrained to be an integer between 0 and 180 degrees. This space contains 162,900 goals.

To generate training data we defined a set of "training goals" that spans the goal space. This smaller set of goals was defined in the same fashion as for the testing set of goals except that the directions in the racecourse are restricted to be only 0, 90, or 180 degrees, yielding a smaller space of 30 goals. To label the

[1] The four operators we chose were *Scale-X*, *Scale-Y*, *Prism-Y*, and *Scale-Keel*. We chose these operators because the results of our earlier work on operator-importance analysis suggested that these are the four most important operators [3].

training data we attempted to find designs for each of the 30 goals starting from each of the four prototypes using the restricted set of operators, and determined which starting point was best.

To generate test data we randomly selected ten "testing goals" from the goal space. We then generated designs starting from each of the four prototypes in the database for each of these testing goals to determine which prototype was best, as well as to determine how much of a loss in course time each incorrect selection would impose. Table 1 compares the results using C4.5 with the other prototype-selection methods. (Since there are four prototypes, we would expect random guessing to get 75% of the test examples wrong.) Figure 2 gives an example of a decision tree output by C4.5.

5: Analysis

In the experiment in the previous section the inductive method (C4.5) performed better than the other methods on both measures of performance. Moreover, we were particularly surprised by how poorly the non-inductive prototype-selection methods (closest goal and smallest initial evaluation) performed—our expectation was that the prototypes chosen by these methods would be close in "design space" to the optimal final design, thus yielding better final designs than starting from the other prototypes.

After studying these results we generated two conjectures for why these two prototype-selection methods did not work well. The first is that the shape of the design space may be such that there is little relationship between the distance between two designs and the ability of the hillclimber to climb from one design to the other. If the space contains "bumps" or "ridges" over which the hillclimber cannot climb, then it might be more important for the initial prototype to be on the "right side" of a bump or a ridge than for it to be close to the optimal point. Our second conjecture was that some of the prototypes in the database may be "bad" prototypes. This could be the case if the hillclimber got stuck at a local (non-global) optimum during the run that produced the prototype. This latter conjecture was supported by the fact that one of the four prototypes was never found to be a good starting point for any of the 30 goals in the training data (not even the goal for which it was supposedly optimal, since it wound up being a local optimum and starting from another prototype yielded a superior result). In a realistic design scenario, when there is no control over the source of a design library, there could easily be "bad" prototypes included. Unlike the non-inductive prototype-selection methods, the inductive methods learn to avoid the bad prototypes.

To test our first conjecture that the closest-goal and smallest-initial-evaluation methods performed poorly because of the "bumps" in the evaluation function, we repeated the experiments using a simplified, "smooth" velocity prediction program, called "RUVPP," that we developed at Rutgers. RUVPP differs from the more complex AHVPP in several respects. To begin with, RUVPP represents yachts as a list of major geometric dimensions such as length, depth, and beam, rather than B-spline surfaces. Furthermore, RUVPP embodies a number of simplifying assumptions about the physics of sailing that are not made in AHVPP. Nevertheless, the simple version, RUVPP, is useful for two reasons: RUVPP is much faster to execute than AHVPP, and RUVPP has fewer of the bumps and ridges that appear in AHVPP. We therefore expect that a hillclimbing search algorithm is less likely to get stuck on the wrong side of a bump or ridge when the simple version, RUVPP, is used as an evaluation function. Table 2 presents the results of experiments comparing the performance of inductively learned prototype-selection rules to the other prototype-selection methods, repeating our earlier experiments, but using RUVPP as the evaluation function, and using forty random test cases instead of just ten.

Because RUVPP is much faster than AHVPP, we conducted additional supporting experiments to test our first conjecture, to see if using a spanning set of goals as training data was significant for our results. In particular, rather than using inductive learning on a set of goals that span the space of possible goals, we also performed experiments where C4.5 was trained on a random sample of goals selected from the same space as the testing data. This was done using ten trials of four-fold cross-validation [16] on a set of forty random goals. Each such trial involves randomly dividing the data into four sets of size ten, using three of the sets for training data and the remaining one as testing. This is repeated four times, using each ten-element set once for testing, and this process was repeated ten times with different random partitionings of the data. Table 3 reports the results of these experiments.

Consistent with our conjecture, the closest-goal and best-initial-evaluation methods both did much better in both cases with the simplified VPP than they did with AHVPP, while C4.5 did about the same as it did before. We believe that because the simplified VPP is much smoother than AHVPP, the hillclimber

Table 1: Comparison of prototype-selection methods when trained on a set of goals that spans the goal space, using AHVPP.

Method	Error Rate	Course-Time Increase (sec)
Inductive Learning	30%	24
Most Frequent Class	70%	47
Random Guessing	75%	62
Best Init Eval	70%	64
Closest Goal	70%	78

is much less likely to get stuck, so that the distance in goal space or the difference in initial evaluation becomes much more relevant when choosing a prototype. In fact, the improvement in the best-initial-evaluation method was so great that it significantly outperformed the inductive method.

To test our second conjecture of why the closest-goal and smallest-initial-evaluation method performed so poorly using AHVPP—that they were unable to avoid the "bad" prototype in the database—we repeated our preceding experiments using the simplified VPP, except that we intentionally put a "bad" prototype into the database. To generate a bad prototype, we started with the Stars and Stripes '87, and added a random number between -0.2 and +0.2 to each of the operator parameters. We then randomly chose one of the four prototypes in the database to replace with the bad prototype (but we left the class label the same). The results of repeating the experiments with the bad prototype in the database are presented in Table 4 for training on goals that span the space, and Table 5 for training on random goals.

```
long-leg <= 90 :
|   windspeed > 10 : Design-1
|   windspeed <= 10 :
|   |   short-leg <= 90 : Design-1
|   |   short-leg > 90 : Design-2
long-leg > 90 :
|   windspeed > 14 : Design-2
|   windspeed <= 14 :
|   |   windspeed <= 10 : Design-4
|   |   windspeed > 10 : Design-4
```

Figure 2: Example of a prototype-selection decision tree generated by C4.5.

Table 2: Comparison of prototype-selection methods when trained on a set of training examples that spans the goal space, using the simplified VPP.

Method	Error Rate	Course-Time Increase (sec)
Best Init Eval	12%	26
Inductive Learning	37%	57
Closest Goal	40%	76
Most Frequent Class	45%	175
Random Guessing	75%	257

Table 3: Comparison of prototype-selection methods when trained and tested on random goals, using cross-validation and the simplified VPP.

Method	Error Rate	Course-Time Increase (sec)
Best Init Eval	12%	26
Inductive Learning	30%	35
Closest Goal	40%	76
Most Frequent Class	45%	175
Random Guessing	75%	257

Table 4: Comparison of prototype-selection methods when trained on a set of goals that span the space, using the simplified VPP, and a "bad" prototype in the database.

Method	Error Rate	Course-Time Increase (sec)
Best Init Eval	10%	80
Inductive Learning	30%	82
Closest Goal	32%	89
Most Frequent Class	45%	171
Random Guessing	75%	348

Table 5: Comparison of prototype-selection methods when trained and tested on a set of random goals, using cross-validation, the simplified VPP, and a "bad" prototype in the database.

Method	Error Rate	Course-Time Increase (sec)
Inductive Learning	19%	38
Best Init Eval	10%	80
Closest Goal	32%	89
Most Frequent Class	45%	171
Random Guessing	75%	348

Consistent with our second conjecture, C4.5's ability to avoid the "bad" prototype improved its performance relative to the other methods. When trained on the spanning goals, C4.5 performed only slightly worse than the smallest-initial-evaluation method. When trained on the random goals, C4.5 performed markedly better than any other method as measured by average course-time increase, although the smallest-initial-evaluation method had a lower error rate. This apparent anomaly can be explained as follows: The "bad" prototype was very bad, so that choosing it even a few times resulted in large increases in average course time. C4.5 never chose the bad prototype. The best-initial-evaluation method occasionally chose the bad prototype, so that even though it chose the best prototype more frequently than C4.5, the few times when it chose the bad prototype worsened its average course-time increase.

6: The cost of learning

One important question to answer is whether the inductive prototype-selection method is worth the considerable "off-line" expense of collecting training data—every training example requires one design run for each design in the prototype library. An alternative, possibly cheaper method would be to take an "on-line" approach: for each new design problem optimize starting from every prototype in the database, and then use whichever of the resulting designs is the best.

If the quality of the final design is extremely important and there is ample CPU time available, this "exhaustive" method is the one to use. On the other hand, if limiting CPU time is important, our inductive learning method becomes cost effective when the computational expense of learning can be amortized over a sufficiently large number of new design goals. More specifically, the inductive prototype-selection method is less expensive than the exhaustive method whenever the number of hillclimbing runs taken by the inductive approach is less than the number of runs taken by the exhaustive approach, i.e., $TP + G < PG$ or

$$G > \frac{T}{1 - \frac{1}{P}}$$

where T is the number of training examples, P is the number of prototypes in the database, and G is the number of new goals for which prototypes need to be selected. In all of the experiments that we performed, there were four prototypes and 30 training examples, so our inductive approach will be less expensive than the exhaustive approach as long as at least 40 out of the more than 150,000 remaining design goals must be attempted.

7: Related work

Cerbone [1] 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. In contrast, our system has a smaller number of prototypes (each of which defines a search space) from which to choose, and it just chooses one of them. Cerbone uses an ad-hoc utility function to combine solution quality and search time when evaluating his learning methods, while we only consider solution quality in this paper. Cerbone also presents two learners that incorporate background knowledge by incorporating the objective function into the learner.

Research on prototype-retrieval strategies for hillclimbing design optimization is reported by Ramachandran et al. [12], who investigated a number of library-based methods for finding starting points for the DPMED iterative parameter-design system. These included a nearest-neighbor method, a curve-fitting method, and a hybrid method. The curve-fitting method is similar in spirit to our inductive learning method. It uses regression to find a function mapping goal parameters to initial design parameters, whereas our approach uses inductive learning to find a set of rules mapping goal parameters to an initial design. Ramachandran compared their retrieval strategies in terms of the numbers of iterations needed to carry out the hillclimbing design-optimization process. They showed that starting points obtained by curve fitting led to fewer iterations than were required when the nearest-neighbor method was used. In contrast to this, our work has evaluated retrieval strategies in terms of the quality of the resulting designs, rather than the number of iterations needed to find them.

Several investigators [6, 15] have developed alternative artificial-intelligence techniques for controlling iterative parameter-design optimization. However, none of these efforts is focused directly on the problem of finding a starting point for iterative design. Likewise, library-retrieval techniques for case-based design

[14, 4] have not been used to initialize an iterative design process.

8: Future work

This paper presents an initial exploration of our inductive approach to the prototype-selection problem, and there are a number of directions for future work. First, the experiments reported here explore the sensitivity of our approach to the nature of the design library, specifically with respect to the quality of the stored designs. It would be helpful to more fully explore the sensitivity of our approach to the design library, for example by studying how our approach scales up as the library size increases. Second, our results are limited to experiments in the domain of racing-yacht design, using course time as the sole evaluation criterion. Evaluating our approach on other types of goals within this domain, such as cost, would lead to a better understanding of our inductive prototype-selection method. We also plan to apply our approach to design in other domains. In particular, there is on-going work at Rutgers on applying the Design Associate to the domain of aircraft-engine nozzle design. We plan to explore the importance of prototype selection for selecting appropriate nozzle architectures.

Further, C4.5 is simply one out of many available inductive-learning methods, and it would be useful to compare our results to those obtained with other learning algorithms (such as neural networks) to see how dependent our results are on the particular inductive method. We also plan to explore how our results can be improved even further through the development of inductive-learning methods that can use background knowledge of the domain (e.g., racing-yacht design) while learning prototype-selection rules. Additionally, learning methods operating on more expressive representations, such as inductive logic programming systems like FOIL [9], may enable going beyond the simple representation of goals used here and handling more complicated goals, such as those involving multiple disciplines. Finally, learning systems operating over continuous value spaces, such as M5 [10], may make it possible to perform *prototype generation* instead of prototype selection. This will involve learning rules to map the goal directly into the parameters that form a prototype, rather than a particular prototype in a database.

References

[1] 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.

[2] 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.

[3] T. Ellman and M. Schwabacher. Abstraction and decomposition in hillclimbing design optimization. Technical Report CAP-TR-14, Department of Computer Science, Rutgers University, January 1993.

[4] J. Kolodner. *Case-Based Reasoning.* Morgan Kaufmann Publishers, San Mateo, CA, 1993.

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

[6] M. Orelup, J. Dixon, P. Cohen, and M. 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.

[7] W. Press, B. Flannery, S. Teukolsky, and W. Vetterling. *Numerical Recipes.* Cambridge University Press, New York, NY, 1986.

[8] J. R. Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[9] J. R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.

[10] J. R. Quinlan. Learning with continuous classes. In *Proceedings of the Australian Joint Conference on Artificial Intelligence*, pages 343–348, Singapore, 1992. World Scientific.

[11] J. Carbonell R. Michalski and T. Mitchell, editors. *Machine Learning, An Artificial Intelligence Approach.* Morgan Kaufmann, Los Altos, CA, 1983.

[12] N. Ramachandran, N. Langrana, L. Steinberg, and V. Jamalabad. Initial design strategies for iterative design. *Research in Engineering Design*, 4:159–169, 1992.

[13] D. Rogers and J. Adams. *Mathematical elements for computer graphics.* McGraw-Hill, 2nd edition, 1990.

[14] K. Sycara and D. Navinchandra. Retrieval strategies in a case-based design system. In C. Tong and D. Sriram, editors, *Artificial Intelligence in Engineering Design (Volume II)*, pages 145 – 164. Academic Press, New York, NY, 1992.

[15] 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.

[16] S. Weiss and C. Kulikowski. *Computer Systems That Learn.* Morgan Kaufmann, Los Altos, CA, 1991.