Optimization by Searching a Tree of Populations

Louis Steinberg

Department of Computer Science
Rutgers University
110 Frelinghuysen Road
Piscataway, NJ 08854
lou@cs.rutgers.edu, 732-445-3581

Khaled Rasheed

Department of Computer Science
Rutgers University
110 Frelinghuysen Road
Piscataway, NJ 08854
krasheed@cs.rutgers.edu, 732-445-6185

Abstract

GAs have been found to be useful in handling many numerical optimization problems. Because of the variability in results inherent in the stochastic nature of GAs, it is common to run a GA several times and take the best of the results. However, it is possible to save a GA's population at some intermediate states and restart from one of these populations instead of from the very beginning. By doing so we generate a tree of populations, where a child population is generated from its parent by running some number of GA iterations. We describe two methods for searching such a tree of populations, one based on Highest Utility First Search (HUFS) and one that proceeds level by level with no backtracking, and give the results of testing them on a real-world optimization task involving conceptual design of supersonic transport aircraft. They both do much better than repeatedly running the GA from the beginning, with HUFS achieving equivalent results in less than half the GA iterations in some situations.

1 Introduction

Genetic Algorithms (GAs) have proven useful for solving numerical optimization problems in number of domains. In such a GA, an individual represents a candidate solution to the problem and the fitness function is derived from the measure of merit being optimized. The best individual created during a run becomes the result of the optimization.

GAs have the feature that if you run one repeatedly on the same problem you will get a range of different results, and these results will vary in quality, i.e in the measure of merit. Even though this variability is often less than with competing methods like gradient descent, it still may be significant. If so, the variability can be further reduced by running the GA a few times on the same problem, and taking the best of the resulting answers as the overall answer.

Current practice is to simply run the entire optimization process several times from beginning to end. Suppose, however, that we ran the GA in stages, stopping after every n iterations and saving the current population. We can view the process of running for n iterations as an operator that generates one population from another. Because of the stochastic nature of this operator, we can apply it to the same "parent" population (that is, go back and restart from the same saved population) many times, and it will generate many different "child" populations. We can also apply the operator to one or more child populations. In this way we can generate a tree of populations.

We can view the optimization process as searching this tree of populations. The basic step is to start with some population and apply the operator (i.e. run the GA for n iterations), to generate one child. When we go back we need not go back to the very beginning, and when we run the GA we need not run to convergence. Rather, each time we do a step we can choose any existing population as our starting point and take one step from there.

This paper will present the results of taking this approach. It will describe two methods for controlling the search, that is, for deciding at each step which population to work from, and will also describe an empirical evaluation of these methods on a problem involving the design of a supersonic airliner. It will be seen that this approach can cut in half the number of iterations needed to achieve a given level of quality, compared to simply rerunning the GA from scratch.

The rest of this section will lay out the context we are

Table 1: Aircraft Parameters to Optimize

No.	*Parameter*
1	exhaust nozzle convergent length(l_c)
2	exhaust nozzle divergent length(l_d)
3	exhaust nozzle external length(l_e)
4	exhaust nozzle radius(r7)
5	engine size
6	wing area
7	wing aspect ratio
8	fuselage taper length
9	effective structural t/c
10	wing sweep over design mach angle
11	wing taper ratio
12	Fuel Annulus Width

working in: the specific problem and GA that we have used for our experiments. The next section will describe the two control methods, one very simple and one based on decision theory. The third section will describe the empirical tests and their results. The fourth section will discuss related work and the last section is our summary and conclusions.

1.1 The Test Problem

Our test problem concerns the conceptual design of supersonic transport aircraft. We summarize it briefly here; it is described in more detail elsewhere [GSS96]. The GA attempts to find a good design for a particular mission by varying twelve of the aircraft conceptual design parameters in Table 1 over a continuous range of values.

An optimizer evaluates candidate designs using a multidisciplinary simulator. In our current implementation, the optimizer's goal is to minimize the takeoff mass of the aircraft, a measure of merit commonly used in the aircraft industry at the conceptual design stage. Takeoff mass is the sum of fuel mass, which provides a rough approximation of the operating cost of the aircraft, and "dry" mass, which provides a rough approximation of the cost of building the aircraft. The fuel mass needed is computed by simulating the aircraft's flight over a specified "mission", which says how far the aircraft flies at each of several altitudes and speeds. Calculating takeoff mass for one candidate design requires about 0.2 CPU seconds on a DEC Alpha 250 4/266 desktop workstation.

The aircraft simulation model used is based on both implicit and explicit assumptions and engineering approximations. Since it is being used by a numerical optimizer rather than a human domain expert, some design parameter sets may correspond to aircraft that violate these assumptions and therefore may not be physically realizable even though the simulator does not detect this fact. We refer to these designs as *infeasible points*. For this reason a set of constraints has been introduced to safeguard the optimization process against such violations. Other constraints enforce requirements such as the requirement that there be room for some specified number of passengers inside the aircraft. The result of optimization must satisfy all of the constraints, as well as having a low takeoff mass.

We generate a range of different problems by varying two parameters: the percentage of the mission that is to be flown at subsonic speeds (to avoid sonic booms over populated areas), and how many passengers the aircraft must accommodate. A problem specification then consists of values for these two numbers.

1.2 The Genetic Algorithm Used

We conducted our investigations in the context of GADO [Ras98, RHG97], a GA that was designed with the goal of being suitable for use in engineering design. It uses new operators and search control strategies that target the domains that typically arise in such applications. GADO has been applied in a variety of optimization tasks that span many fields, and has demonstrated a great deal of robustness and efficiency relative to competing methods.

In GADO, each individual in the GA population represents a parametric description of an artifact, such as an aircraft or a missile. All parameters take on values in known continuous ranges. Floating point representation is used. The fitness of each individual is based on the sum of a proper measure of merit computed by a simulator or some analysis code (such as the takeoff mass of an aircraft), and a penalty function that is a function of the number and magnitude of constraint violations. A steady state GA model is used, in which operators are applied to two parents selected from the elements of the population via some selection scheme, one offspring point is produced, then an existing point in the population is replaced by the newly generated point via some replacement strategy. Here selection was performed by rank because of the wide range of fitness values caused by the use of a penalty function. The replacement strategy used here is a crowding technique, which takes into consideration both the fitness and the proximity of the points in the GA population. The population size is an external parameter with a default value of 10 times the dimension of the search space. For the aircraft problem we used the default, giving 120 individuals in the population. The initial population is generated by randomly generating many individuals and using the replacement strategy to keep the population at 120.

Several crossover and mutation operators are used, most of which were designed specifically for numerical optimization problems of this type. GADO also uses a search-control method that saves time by avoiding the full evaluation of points that are unlikely to correspond to good designs. The GA stops when either the maximum number of evaluations has been exhausted or the population loses diversity and practically converges to a single point in the search space.

GADO calls the simulator to evaluate a candidate about $12,000$ times in order to converge on an answer to the aircraft design problem, including $3,600$ calls in the initialization phase. We have chosen to divide this into five chunks, a first group of $4,000$ iterations to get past the initialization, and four more chunks of $2,000$ iterations each. (The $2,000$ iteration chunk size was based on trading off the need for a tree that was deep enough to be realistic with the desire to make the tree as small as possible to simplify the experimentation.) Thus, the tree to be searched has a root representing the initial state with problem specifications but no individuals generated yet, and five levels whose nodes are populations after, respectively, 4, 6, 8 10, and 12 thousand calls to the evaluator.

2 Searching the Tree of Populations

Searching a tree is one of the classic tasks in Artificial Intelligence, and in Computer Science in general. However, the tree search problem here is somewhat different from the problems we normally think of.

- The tree is quite shallow, with a depth of at most 5, but the branching factor is combinatorially large - it is the number of different populations we could possibly arrive at in $2,000$ (or $4,000$) iterations.

- the only way we have to generate children from a node is a stochastic operator that, each time it is applied, randomly selects which of the possible children it will return. Furthermore, this operator takes five to ten minutes to apply, so we can afford a fairly large amount of computing per operator application for reasoning about control of the search.

- While we care about the total time taken, and thus the total number of operator applications, we do not care about how long the path in the tree is from root to the solution. This is in contrast to planning problems where the path in the the tree represents a sequence of steps to be carried out so shorter paths are preferred.

Because of these differences, some search algorithms, e.g. A^*, are irrelevant (since we do not want to optimize path length) and/or impossible (since we cannot practically generate all children of a node). Furthermore, finding the absolute global optimum design almost always takes more computing time than we can afford, so we need to trade off computing time for result quality. One search control method that applies naturally in this situation is Highest Utility First Search (HUFS) [SHD98]. In this section we will first discuss the notion of utility in general and the particular formulation we used, then we will discuss HUFS, and then we will describe a much simpler search method, Waterfall, that we also tested.

2.1 Utility and Satisficing

The notion of utility comes originally from the field of decision theory. A decision results in some outcome, and the utility of an outcome is the overall net benefit of this outcome. The utility of an action is the average utility of the outcomes that may result from the action, weighted by their probability of occuring. In the literature on time-bounded computing. e.g., [RW91], the utility of computing some result r is seen as a function $U(r,t)$ of the result itself and the time, t, at which it becomes available. By reasoning about the expected utility of alternative actions, a control method can handle the tradeoff between compute time and result quality.

In much of the literature, U is further assumed to be of the form

$$U(r,t) = I(r) - C(t)$$

where $I(r)$ is some intrinsic value in having r, and $C(t)$ represents a "time cost", usually a linear function of t, i.e. a constant cost per unit of time. For our purposes, however, this formulation has a major drawback in that it does not easily handle deadlines. In most real world engineering situations, there is some deadline by which a design simply must be produced; a design that is produced later, no matter how good it is intrinsically, is worth nothing. We believe that most real world engineering design also has a satisficing character. The designer has in mind some goal level of design quality which is "good enough", and the primary aim of the designer is to produce a design that is good enough (meets or exceeds the goal level of quality) and to do so soon enough (before the deadline). Thus, we assume there is some deadline, and formulate the utility of a result as a simple threshold function on its quality and the current time:

$$U_r(Q(r), t) \;=\; \begin{array}{l} 1 \text{ if } \mathrm{Q(r) \leq Q_g} \text{ and } \mathrm{t \leq t_g} \\ 0 \text{ otherwise} \end{array}$$

where Q is some quality metric (e.g., takeoff mass), where Q_g is the desired quality and t_g is time the deadline occurs. (Note that lower Q is better.) Then the utility of an action becomes simply the probability that it will result in a design with quality of Q_g or better by time t_g. This is of course also a simplification of the real world, but we believe it is a better model than using a fixed cost per unit time.

2.2 HUFS

HUFS works by estimating, for each possible parent, the expected utility of the design process that would result if the search were restricted to the given parent and its descendants. In the following we will describe how these expected utilities are estimated, and then will describe how HUFS uses these estimates to do its search. We start by discussing the idea of a *Child Score Distribution*.

2.2.1 Scores and Child Score Distributions

We assume that we have some heuristic evaluation function $S(a)$ that assigns a numerical score to a population a. The score of a population is an estimate of how good a result we will get if we use it as our starting point. For simplicity we have defined $S(a)$ to be the quality metric of the best individual result in a. That is, viewing population a as a set of candidate results, since lower Q is better,

$$S(a) = \min_{r \in a} Q(r)$$

This means that if we stop searching and take the best result r' in a as our answer to the optimization problem, then $Q(r') = S(a)$. So, our search succeeds if we find a population with $S(a) \leq Q_g$ by time t_g.

If we generate a child (i.e., run the GA a specified number of iterations and generate a new population) and calculate its score, the value we get depends to some extent on the randomized choices made by the GA while generating the child, so we can treat the score as a random variable. Since generating a child does not change the parent, the score of one child has no effect on the score we will get if we generate another child from the same parent. Therefore for a given parent, a, we can view the scores of its children as independent variables with identical distributions. We define the *Child Score Distribution*, G_a of a parent, a, to be this probability distribution of its childrens' scores. That is,

$$G_a(s) = P(S(b) = s | b \in \hat{C}(a))$$

where $\hat{C}(a)$ is the set of populations that might be generated as children of a.

We assume that G_a is a normal distribution, and that the mean and standard deviation of G_a are themselves randomly drawn, respectively, from normal distributions M and D. We assume the standard deviations of M and D are constants and that their means are linear functions of the score $S(a)$. That is, we assume

$$M(\mu) = P(\mathrm{mean(G_a)} = \mu) = \mathrm{Z}(\mu, \mathrm{x} * \mathrm{S(a)} + \mathrm{y}, \mathrm{d})$$

where $Z(\mu, x * S(a) + y, d)$ is the Gaussian "bell curve" function with mean $x * S(a) + y$ and standard deviation d, calculated at point μ, and where x, y, and d are constants. We make a similar assumption about D. (In all cases, the distributions are truncated so that the probability of, e.g., $\mu \leq 0$ is 0.) The constants are estimated for each level of the tree by some initial exploration. This gives us an a priori estimate of G_a based on $S(a)$, and we update the estimate as we generate children from a and thus sample the actual distribution. The assumptions give only crude approximations to the true distributions, but as will be seen below the approximations are good enough to allow HUFS to perform well in practice.

This is similar to the approach to estimating child score distributions in [SHD98]; please see that paper for a fuller discussion.

2.2.2 Estimating Utilities

Next, we will describe how the Child Score Distributions are used to estimate the utility of searching under a given parent. In doing so, we will use the following notation:

- As above, $\hat{C}(a)$ is the set of child populations that *can* be generated, while $C(a)$ is the set of children that *have* been generated already.
- $S_{\min}(a) = \min_{c \in C(a)} S(c)$, i.e. it is the minimum score of any child of a. If a has no children, $S_{\min}(a) = \infty$.
- τ is the time needed to generate one child.
- $U_r(q, t)$ is the expected utility of having a result with quality q by time t

- $U_p(a,t)$ is the utility of searching under parent a starting the search at time t. Note that while $U_r(q,t)$ is either 0 or 1, $U_p(a,t)$ is 0, 1, or any number in between.

- $U_p(a,t|C(a))$ is the expected value of $U_p(a,t)$ given some condition $C(a)$. E.g., $U_p(a,t|S(a)=s)$ is the expected value of $U_p(a,t)$ given that a's score is s.

- We number levels in the tree upward from the lowest level, level 0, to the root (level 5), and use superscripts to denote the level. So $U_p^1(a,t)|S(a)=s)$ is the utility of searching under a population a at level 1 whose score is s.

The utility of searching under a parent of course depends on the search algorithm used, and the utility estimates themselves are part of the search algorithm. We simplify this self-referential problem by assuming a restricted search algorithm in our utility analysis: to search under parent a at level i in the tree, generate children of a until we find a child c whose estimated U_p is larger than that of the a, that is, until

$$U_p^{i-1}(c,t) > U_p^i(a,t)$$

where t is the current time. then apply this algorithm recursively to c. Stop when you find a child whose score is less than (i.e. better than) Q_g.

This restricted version ignores the possibility that after we switch to c and generate some of *its* children, we may revise our estimated utilities and decide that some sibling of c, or perhaps even a itself, has a higher utility. In that case the actual HUFS algorithm would generate the next child from the parent with the currently-highest U_p, while the restricted algorithm would not.

To estimate $U_p^i(a,t)$, where a is the parent we will work recursively on both time and level number. There are two base cases.

- If $t_g < t$, i.e. the deadline has passed, all utilities are 0.

$$\mathrm{if\ t_g < t\ then\ U_p(a,t) = 0}$$

- If a is on level 0, we do not generate any further children, so all we can do is take the best result in a as our answer, and

$$\begin{aligned} U_p^0(a,t) &= U_r(S(a),t) \\ &= \mathrm{1\ if\ S(a) \leq Q_g\ and\ t \leq t_g,} \\ &\quad \mathrm{else\ 0} \end{aligned}$$

In general, $U_p^i(a,t)$ depends on $S(a)$, $G(a)$, and, if a has any children, on $\min_{c\in C(a)} S(c)$.

- If we choose to stop and use the best result in a as our answer, then as above

$$U_p^i(a,t) = U_r(S(a),t)$$

- If we choose to switch to generating children from a child of a, we will choose the child with the best utility, and the utility of our search becomes that child's utility. But, by the restrictions on the algorithm discussed above, we have not generated any children from any child of a, so all we know about a's children are their scores. The best child is then one with the lowest score, so

$$U_p^i(a,t) = U_p^{i-1}(c,t|S(c) = S_{\min}(a))$$

Note that it takes no time to switch so we still use time t.

- If we choose to generate another child from a, it will be available at time $t+\tau$. It will have score s with probability $G_a(s)$, but it will only change $U_p^i(a,t+\tau)$ if it is better than the best child of a we already have, i.e. has a lower score, so the expected utility is

$$\begin{aligned} & U_p^i(a,t) = \\ & \int_0^{S_{\min}(a)} G_a(s) U_p^{i-1}(c,t+\tau|S(c)=s) ds \\ & \quad + \quad U_p^{i-1}(c,t+\tau|S(c)=S_{\min}(a),t) \\ & \qquad * \int_{S_{\min}(a)}^{\infty} G_a(s) ds \end{aligned}$$

In fact, we will choose the one of these three alternatives with maximum utility, so $U_p^i(a,t)$ is the maximum of these three possible values. Note that where we recur it is on a lower level or a later time, so the recursion terminates.

2.2.3 The HUFS algorithm

Given the discussion above, HUFS can be described as simple best-first search where best is defined as having the largest $U_p^i(a,t)$. HUFS starts with a tree consisting of just a root node, representing the problem specification. The algorithm is:

- Repeatedly do the following:
  - Find the node in the tree with the largest $U_p^i(a,t)$, where t is the current time. Call this node a'.
  - If $U_p^i(a',t) = U_r(S(a'),t)$, stop and return a' as our answer, since that is the highest utility use of a'.
  - Otherwise, generate one child from a' and use the score of this new child to update our estimate of $G_{a'}$

2.3 Waterfall

As a check on whether we needed the complexity of HUFS, we also tried a very simple search algorithm, the Waterfall algorithm from [SHD98]. This algorithm works from the top down. At each level it takes the population with the lowest (best) score and generates a fixed number of children from it, then it chooses the best of those children and repeats.

3 Empirical Test

The empirical test was run on the aircraft optimization problem described above. Rather than use CPU time for t and τ we counted iterations of GADO. The implementation uses C and LISP, and runs on SUN workstations. Generating a population takes 5-10 minutes and running HUFS takes up to a minute or so per population.

We first created 10 random problems (combinations of number of passengers and percent supersonic) for calibration. On each problem we generated a population at each level of the tree and generated 10 children for each of these. From the scores of these populations we estimated the parameters for the a priori estimates of $G(a)$ as a function of $S(a)$.

Next we created 4 more problems for testing. We then ran HUFS and Waterfall 12 times on each and also ran plain GADO 12 times on each problem, for each of several different t values. Since repeated runs of GADO are independent, we simulated the process of repeating GADO by repeatedly randomly choosing a run from the 12 we had done on a given problem. We did this 1000 times for each combination of problem and t.

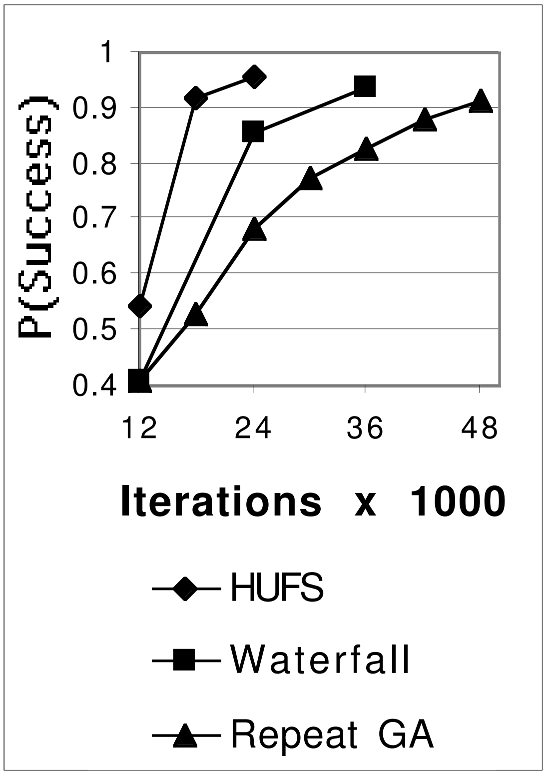

Figure 1: P(Success) averaged over 4 test problems

The Q_g was set for each problem so that repeating GADO would take about 48000 iterations to have a 90% chance of success, i.e. of achieving Q_g.

Results are shown in Figure 1. It can be seen that if it is worthwhile doing extra GADO iterations (beyond the 12,000 needed for one full run of GADO) in order to improve the probability of success, then doing a tree search guided by HUFS can get the same probability of success in half the GADO iterations or less compared to simply repeating GADO over and over.

Waterfall's performance is also much better than that of repeated GADO, but not as good as HUFS. On the other hand, HUFS is much more complex.

4 Related Work

Several research efforts have applied genetic algorithms to engineering optimization and search problems in a variety of domains, including control system design [KK96], architectural and civil engineering design [GKS97, Ros97], VLSI design [LT93], mechanical design [CJ96] and aircraft design [OYN97]. Deb [DG97, Deb97] developed a GA called GeneAS for engineering design optimization with mixed variables (both discrete and continuous). He demonstrated the merit of his GA in the domain of mechanical component design. Powell [Pow90, Ton88] has built a module called Inter-GEN, part of the ENGINEOUS system [TPG92]. It contains a genetic algorithm and a numerical optimizer, and uses a rule-based expert system to decide when to switch between the two. Powell tested his system on a realistic design task (jet engine design). Combining GAs and knowledge-based systems was also done in [RMB96].

Several research efforts have focused on preventing premature convergence in GA search. An important class of methods are replacement strategies that take into consideration other factors in addition to fitness (such as preserving diversity in the population for example). Each such strategy is called a *crowding heuristic* and the reader is referred to [Mah95] for a detailed discussion of these methods. Another class of methods focus on carefully choosing the population size and the reader is referred for example to [HCPGM97].

Several research efforts have used utility based decision making to improve design optimization, e.g., [RW91] and [Etz91]. [SHD98] is the closest of these to the work described here. It uses the HUFS algorithm to control search in a tree whose levels are problem solutions at different levels of abstraction, and it uses a constant cost per unit time in its formulation of utility, but otherwise is quite similar in approach to this paper.

To the best of our knowledge, no research efforts have attempted to combine utility based decision making with GA search in a way similar to the proposed approach, nor has any previous work looked at searching the space of GA populations.

5 Conclusions

We have demonstrated that, in a realistic engineering optimization problem, it can be useful to think of a GA as generating a virtual tree of populations, and to explicitly search this tree, either with HUFS or Waterfall as the search control method. These results clearly need to be replicated on additional problems and with other GAs. We believe that the idea of searching the space of populations, rather just using the GA's operators to generate a sequence of populations, will be a very fruitful one to explore.

We have also added a data point regarding the generality of HUFS, showing that is is useful even when the tree being searched is not made up of design abstraction levels. Rather, HUFS seems to apply more generally to trees with small, limited depth, combinatorially large branching factors, and an operator that generates random children.

Acknowledgements

The work presented here is part of the "Hypercomputing & Design" (HPCD) project, and benefited greatly from both the intellectual and software environments provided by our colleagues on that project. Thanks also to Prof. Robert Berk.

This work is supported (partly) by ARPA under contract DABT-63-93-C-0064 and by NSF under Grant Number DMI-9813194. The content of the information herein does not necessarily reflect the position of the Government and official endorsement should not be inferred.

References

[CJ96] C. D. Chapman and M. J. Jakiela. Genetic algorithm-based structural topology design with compliance and topology simplification considerations. *Journal of Mechanical Design*, 118(1):89–98, 1996.

[Deb97] Kalyanmoy Deb. Geneas: A robust optimal design technique for mechanical component design. In *Evolutionary Algorithms in Engineering Applications*, pages 497–514. Springer-Verlag, 1997.

[DG97] Kalyanmoy Deb and Mayank Goyal. Optimizing engineering designs using a combined genetic search. In *Proceedings of the Seventh International Conference on Genetic Algorithms*. Morgan Kaufmann, 1997.

[Etz91] Oren Etzioni. Embedding decision-analytic control in a learning architecture. *Artificial Intelligence*, 49:129–159, 1991.

[GKS97] John S. Gero, Vladimir A. Kazakov, and Thorsten Schinier. Genetic engineering and design problems. In *Evolutionary Algorithms in Engineering Applications*, pages 47–68. Springer-Verlag, 1997.

[GSS96] Andrew Gelsey, M. Schwabacher, and Don Smith. Using modeling knowledge to guide design space search. In *Fourth International Conference on Artificial Intelligence in Design '96*, 1996.

[HCPGM97] Harik, Cantu-Paz, Goldberg, and Miller. The gambler's ruin problem, genetic algorithms, and the sizing of populations. In *IEEECEP: Proceedings of The IEEE Conference on Evolutionary Computation, IEEE World Congress on Computational Intelligence*, 1997.

[KK96] Sourav Kundu and Seiichi Kawata. AI in control system design using a new paradigm for design representation. In *Fourth International Conference on Artificial Intelligence in Design '96*, 1996.

[LT93] Jens Lienig and K. Thulasiraman. A genetic algorithm for channel routing in VLSI circuits. *Evolutionary Computation*, 1(4):293–311, 1993.

[Mah95] Samir Mahfoud. A comparison of parallel and sequential niching methods. In *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 136–143. Morgan Kaufmann, July 1995.

[OYN97] S. Obayashi, Y. Yamaguchi, and T. Nakamura. Multiobjective genetic algorithm for multidisciplinary design of transonic wing platform. *Journal of Aircraft*, 34(5):690–693, 1997.

[Pow90] David Powell. Inter-GEN: A hybrid approach to engineering design optimization. Technical report, Rensselaer Polytechnic Institute Department of Computer Science, December 1990. Ph.D. Thesis.

[Ras98] Khaled Rasheed. GADO: A genetic algorithm for continuous design optimization. Technical Report DCS-TR-352, Department of Computer Science, Rutgers University, New Brunswick, NJ, January 1998. Ph.D. Thesis, http://www.cs.rutgers.edu/∼krasheed/thesis.ps.

[RHG97] Khaled Rasheed, Haym Hirsh, and Andrew Gelsey. A genetic algorithm for continuous design space search. *Artificial Intelligence in Engineering*, 11(3):295–305, 1997. Elsevier Science Ltd.

[RMB96] James L. Rogers, Collin M. McCulley, and Christina L. Bloebaum. Integrating a genetic algorithm into a knowledge based system for ordering complex design processes. In *Fourth International Conference on Artificial Intelligence in Design '96*, 1996.

[Ros97] M. A. Rosenman. The generation of form using an evolutionary approach. In *Evolutionary Algorithms in Engineering Applications*, pages 69–86. Springer-Verlag, 1997.

[RW91] Stuart Russell and Eric Wefald. *Do the Right Thing*. MIT Press, 1991.

[SHD98] Louis Steinberg, J. Storrs Hall, and Brian Davison. Highest utility first search across multiple levels of stochastic design. In *Proceedings of the Fifteenth National Conference on AI*, pages 477–484, Madison, July 1998.

[Ton88] 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.

[TPG92] Siu Shing Tong, David Powell, and Sanjay Goel. Integration of artificial intelligence and numerical optimization techniques for the design of complex aerospace systems. In *1992 Aerospace Design Conference*, Irvine, CA, February 1992. AIAA-92-1189.