Proceedings of the American Control Conference
Albuquerque, New Mexico June 1997
0-7803-3832-4/97/$10.00 © 1997 AACC

Genetic Algorithms and Simulated Annealing for Robustness Analysis

Xiaoyun Zhu[1] **Yun Huang**[2] **John Doyle**[3]
California Institute of Technology 116-81
Pasadena, CA 91125

Abstract

Genetic algorithms (GAs) and simulated annealing (SA) have been promoted as useful, general tools for nonlinear optimization. This paper explores their use in robustness analysis with real parameter variations, a known NP hard problem which would appear to be ideally suited to demonstrate the power of GAs and SA. Numerical experiment results show convincingly that they turn out to be poorer than existing branch and bound (B&B) approaches. While this may appear to shed doubt on some of the hype surrounding these stochastic optimization techniques, we find that they do have attractive features, which are also demonstrated in this study. For example, both GAs and SA are almost trivial to understand and program, so they require essentially no expertise, in sharp contrast to the B&B methods. They may be suitable for problems where programming effort is much more important than running time or the quality of the answer. Robustness analysis for engineering problems is not the best candidate in this respect, but it does provide an interesting test case for the evaluation of GAs and SA. A simple hill climbing (HC) algorithm is also studied for comparison.

1. A Simple Robustness Analysis Problem

Robustness analysis can be naturally formulated as a structured singular value, or μ, problem. Consider a discrete-time LTI system with real parametric uncertainty

$$x(k+1) = A(\delta)x(k), \tag{1}$$

where δ is a vector of parameters entering rationally such that $A(\delta) = F_l(M,\Delta)$, where $F_l(M,\Delta)$ is an LFT on $\mathbf{\Delta} := \{diag\{\delta_1 I_{k_1}, \cdots, \delta_n I_{k_n}\} : \delta_i \in R,\ k_i \in Z^+\}$ with $M = \begin{bmatrix} M_{11} & M_{12} \\ M_{21} & M_{22} \end{bmatrix}$. Let $\mathbf{B\Delta} := \{\Delta \in \mathbf{\Delta} : \|\Delta\| \le 1\}$. Then the above LFT is well-posed for all $\Delta \in \mathbf{B\Delta}$ iff $\mu_\Delta(M_{22}) < 1$.

Define $\Delta_c = \{\frac{1}{z}I, z \in C\}$, and $\tilde{\Delta} = diag\{\Delta_c, \Delta\}$, then the robust stability is guaranteed iff $I - \tilde{\Delta}M$ is nonsingular for all $\tilde{\Delta} \in \mathbf{B}\tilde{\mathbf{\Delta}}$ iff $\mu_{\tilde{\Delta}}(M) < 1$. The computation of this mixed μ problem is known to be NP hard, which is generally taken to mean that it cannot be computed in polynomial time for the worst case. By the main loop theorem one necessary condition for $\mu_{\tilde{\Delta}}(M) < 1$ is $\mu_\Delta(M_{22}) < 1$. The computation of $\mu_\Delta(M_{22})$ is called "μ on a box" problem([4]), which can also be regarded as a special case of the mixed μ problem.

Let $\bar{\lambda}_r(\Delta M)$[1] denote the maximum real eigenvalue of ΔM. Then

$$\mu_\Delta(M) = \max_{\Delta \in \mathbf{B\Delta}} \bar{\lambda}_r(\Delta M). \tag{2}$$

unless there is no real eigenvalue when $\bar{\lambda}_r(\Delta M) = 0.$. In addition to Δ being real, we also assume M is real and δ is non-repeated($k_i = 1, \forall i$). Then it can be shown that the maximum must be achieved on the vertices of the parameter space $\mathbf{B\Delta}$, which is not true when any of the three assumptions is violated. As we will see later, taking advantage of this fact greatly simplifies the implementation of the algorithms. This is perhaps the simplest NP hard problem that arises in robustness analysis, and would be the first step in more sophisticated analysis of robust stability or performance of linear systems, or bifurcation analysis of parametrically dependent nonlinear systems.

The most straightforward way to get the maximum is to evaluate the function on all the vertices, but the computation time required goes up exponentially with the dimension. For n = 4, 16 and 64, the time needed to check all the 2^n vertices on a typical workstation is approximately 0.01 seconds, 5 minutes, and 9×10^{10} years respectively. Obviously the computation cost for large problems is devastating and the known NP hardness supports this observation. Fortunately, there are upper and lower bounds with polynomial time algorithms. Descriptions of these algorithms are presented in [4] and [5]. Unfortunately, these cannot be guaranteed to be tight bounds because even the *approximation* problem for $\mu_\Delta(M)$ is NP hard. That is, computing $\mu_\Delta(M)$ to within a constant factor of the optimal is NP hard. So it is a "hard" NP hard problem from the point of view of computational complexity. Nevertheless, we can cheaply bound $\mu_\Delta(M)$ and also refine the bounds by branch and bound.

2. Branch And Bound

Branch and bound (B&B) is a general technique for those optimization problems whose bounds depend on the domain of the problem. It has been proven to be quite useful in refining the bounds for problems such as the one considered here [4]. Our experience has shown that the average quality of the bounds themselves is critical. The intuition behind this is that there are occasionally bad problems where the bounds are poor, but that branching creates new problems where the bounds are good. For this to be successful, the bounds must be good on average so that the branching process moves bad problems into easy problems. Interestingly, it has seemed less critical that the branching scheme be particularly clever.

These points can be readily illustrated on the problem we are considering. We used a naive branching scheme which consisted of a simple heuristic to choose a branch variable, followed by splitting that variable into 2 equal parts, creating 2 new independent problems on which the bounds are computed. (A more sophisticated algorithm would optimize both the variable chosen and the location of the cut.) The global lower bound is the maximum over all the local lower bounds and the global upper bound is the maximum over all the local upper bounds. A branch can be pruned when its local upper bound is lower than the global lower bound. It is essential that branches be pruned effectively to avoid exponential growth. Just cutting each variable once produces 2^n subproblems unless some of them are pruned in the process.

As our first numerical experiment, we computed upper and lower bounds for matrices of size 4, 8, 16, 32, and 64, with

[1]xyzhu@cds.caltech.edu, http://cds.caltech.edu/~xyzhu
[2]yunhuang@cds.caltech.edu
[3]doyle@cds.caltech.edu, http://cds.caltech.edu/~doyle
[1]We will use M instead of M_{22} later on to simplify the notation.

50 matrices of each size. The elements of the matrices were zero mean normally distributed psuedorandomly generated floating point numbers. The quality of the bounds is good on average. Even for 64 x 64 matrices, the normalized errors are within .2. However, there are occasionally quite poor bounds for any size problem. So the B&B algorithm was used to refine the bounds. Interested readers are referred to [7] for more details about the algorithms and the results of this test. Figure 1 focuses on the worst problems for each problem size and plots the number of branches required to achieve a given error in the bound, for 15%, 10%, 5%, 2% and 1%. Since this is a log-log scale, straight lines indicate polynomial growth, and flat lines indicate no growth.

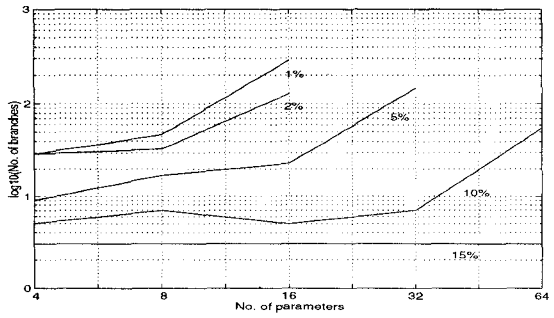

Figure 1: No. of branches versus problem size for various tolerances, for the worst problem out of 50 random matrices in each size. The vertical axis is on a $\log_{10}$ scale, while the horizontal axis is on a $\log_2$ scale.

It is not clear what the growth rates are for small percentages, as the 2% and 1% cases where only done up to size 16. Beyond that, the computation time was too great (several hours per problem) to be practical on individual workstations. Note that the worst problem required exactly 3 branches for each problem size to reach 15%, and that 10% was easily achieved for all problem sizes. This observation supports the notion that branch and bound can easily take the worst-case problems and get them to roughly the level that the bounds achieve on average, but not much better. Due to NP hardness, there must exist truly bad examples where even very clever branch and bound fails, but these seem so rare that they are very unlikely to be encountered in practice. However, the claimed success of some stochastic approaches like genetic algorithms and simulated annealing on many optimization problems makes us hope that they might be better ways to compute the lower bound of this problem.

3. Stochastic Optimization Techniques

Genetic algorithms (GA) and simulated annealing (SA) are two types of stochastic search techniques which are modeled on processes found in nature (natural evolution and thermodynamics respectively). Both have been studied extensively and applied to numerous domains, e.g. VLSI layout designing, machine learning, etc. The most direct application has been function optimization. Optimal solutions of combinatorial optimization problems such as the traveling salesman problem can, in principle, be searched evolutionarily and rapidly. Ackley's paper in [1] provides a comprehensive assessment of the relative performance of these stochastic approaches on a variety of the optimization problems.

3.1. Genetic Algorithm (GA)

Genetic Algorithms, originated by John H. Holland in the 1960s, are based on the principles of natural selection and adaptation and are claimed to be able to explore good solutions quickly on large and complicated search spaces. The power of the algorithms comes from the mechanism of evolution, which allows searching through a huge number of possibilities for solutions. The simplicity of the representations and operations in the GAs is another feature to make them so popular. Bit strings are used in Holland's work to encode candidate solutions to the problems, which are computational analogs to "chromosomes" of a species in nature, while each bit in the string is an analogue to the "gene". A fitness function is evaluated on these chromosomes, then genetic operators transform the parent chromosomes to their offspring according to their fitness rating. Commonly used genetic operators are composed of selection, crossover and mutation.

A lot of research effort has been focused on finding the interior mechanisms that make GAs work. Theoretical explanations include Holland and Godenberg's Schema Theorem and Building-Block Hypothesis. The theorem points out that a GA works because of its capability to evaluate the fitness of schemas by evaluating the fitness of the bit strings in the population. As the population evolves, the search is biased to focus more and more intensively on instances of fit schemas, and crossover makes it possible to explore fit higher-order schemas by combining fit lower-order schemas. Mutation creates new schemas to prevent a permanent loss of information in the population. Related work includes Bethke's Walsh Analysis to characterize the difficulty of functions for GAs to optimize.

The implementation of a GA involves some preparatory stages. First, an objective function has to be determined. In our problem we can define $f(\Delta) = \bar{\lambda}_r(\Delta M)$ and the objective is to find $\max_{\Delta \in \mathbf{B\Delta}} f(\Delta)$. Second, we need a nice representation for Δ. Exploiting the fact that all the candidate solutions are vertices of the $\mathbf{B\Delta}$ greatly simplifies this step since we can use the bit strings representation. For problems of size n, an n-bit string encodes a vertex with 0's corresponding to -1's. For more general μ problems, this is not the case, and the GA would have to search the whole space. Continuous versions of GA have been proposed for dealing with this kind of situation but we want to see first how the GA performs on the simpler case. (The above function and representation apply to the SA and HC as well.)

The algorithm we use here is a simple version based on [3] which consists of the following steps:

1. Generate an initial population of N random n-bit strings (chromosomes). Uniform probability is assumed.
2. $\bar{\lambda}_r(\Delta M)$ is evaluated on each chromosome Δ in the current population and the fitness function $F(\Delta)$ is computed.
3. Reproduce a new generation. A pair of parent strings are chosen from the current population. The probability of the particular instance Δ_1 being selected is $\frac{F(\Delta_1)}{\sum_\Delta F(\Delta)}$, which gives the fit strings a better chance to reproduce. With a crossover rate of p_c, these two chosen parents exchange part of their strings at a random position to form two offspring. Then each bit of the new strings is mutated with a probability p_m. Repeat this procedure until N new offspring are generated.
4. Replace the current population with their offspring, and go back to step 2.

The fitness scaling technique suggested in [2] is employed to compute the fitness function $F(\Delta)$ in order to prevent premature convergence at the beginning of a run and to speed up convergence late in a run. There are three parameters to choose: population size N, p_c and p_m. We pick N to be equal to the problem size n. It makes the computation cost of evolving one generation in GA approximately the same as that of branching once in B&B. The choice of p_c and p_m has a fairly big effect on the performance of the algorithm. Our simulation shows that a single value of p_m would not work for all problem sizes. Higher p_m (0.025) works better for $n = 8$ while lower p_m (0.005) works better for $n = 64$. This suggests that we choose p_m as a decreasing function of the problem size. On the other hand, a single value of $p_c = 0.5$ seems to work fine for all sizes. Since our focus here is not on finding the best parameters for GAs, we will use the best setting we can find for each size when we make comparisons.

3.2. Simulated Annealing (SA)

Simulated Annealing was first proposed by Kirkpatrick *et al* in 1983 who established a strong relation between statistical mechanics and combinatorial optimization problems. All candidate solutions to the problem are modeled as possible configurations of a thermal system. Thus the parameter space S becomes the space of all configurations. The energy E of the system, analogue of the objective function, depends on the current configuration. The minimum energy configuration corresponds to the optimal solution.[2] By the Boltzmann distribution, if a system is in thermal equilibrium at a given temperature T, then,

$$\pi_T(u) = \frac{e^{\frac{-E(u)}{kT}}}{\sum_{v\in S} e^{\frac{-E(v)}{kT}}}, \tag{3}$$

where $\pi_T(u)$ is the probability of the system being in a certain configuration u, and k is the Boltzmann's constant([1]).

This principle shows that at high temperatures all configurations are almost equally likely to be the current configuration while at low temperatures low energy configurations are predominant. However, to obtain the configurations with near-lowest energy, simply decreasing the temperature is not sufficient. This is why an annealing process is applied in thermodynamics, in which the temperature is first elevated, and then gradually reduced, spending enough time at each temperature to let the system reach the thermal equilibrium. In the optimization problem this procedure is simulated. An algorithm called *Metropolis loop* is run at each fixed temperature T for a certain amount of time to approximate the equilibrium state([6]). The essence of this algorithm is it accepts a "bad" point occasionally to escape from a local minimum. And the probability of such an acceptance drops when the temperature goes down.

It is easy to apply SA to our problem. The algorithm we use is as follows:

1. Initialization: $T = T_{max}$. Generate a vertex Δ_c at random, evaluate the function $f_c = f(\Delta_c)$.

2. *Metropolis loop* (Repeat L times): Randomly pick one bit in Δ_c and mutate it, get a new vertex Δ_n, evaluate the function $f_n = f(\Delta_n)$. Let $\Delta_c \leftarrow \Delta_n$ with probability $e^{\frac{f_n - f_c}{T}}$.

[2]This is true for minimization problems only. A slight variation is needed for maximization.

3. Decrease T: $T = \gamma T$. If $T \geq T_{min}$, go to step 2; otherwise, stop.

The control parameters we need to determine are: initial temperature T_{max}, minimum temperature T_{min}, decay rate γ, and length of the *Metropolis loop* L. To make the comparison easy, we choose L to be the problem size n. Then each execution of the *Metropolis loop* takes n function evaluations so that it can be considered as one step which is computationally comparable to one generation in GAs. Different combinations of the other 3 parameters have been tested and the final values used for testing SA on general random problems are: $T_{max} = 100, T_{min} = 0.001, \gamma = 0.8$.

4. Empirical Comparisons

In this section, the GA and SA described above are tested on the elementary case of the μ computation defined in section 1. A simple hill climbing (HC) algorithm is also applied for comparison. We consider the number of generations in GAs and the number of steps in SA as equal measures of computation cost as the number of branching in B&B. [3]

4.1. Rank-one Problems

Our first numerical test focused on a special class of matrices: rank-one matrices. Without loss of generality, consider $M = aa^T$ with $a = [a_1\ a_2\ \cdots\ a_n]^T \in R^n$ and $\|a\| = 1$. Then

$$\bar{\lambda}_r(\Delta M) = \sum_{i=1}^{n} a_i^2 \delta_i. \tag{4}$$

So $\mu_\Delta(M) = 1$ with the maximum achieved at the vertex where all $\delta_i = 1$. In this case the problem can be solved analytically in linear time, and the B&B is guaranteed to solve it in linear time without any branching. It's also trivial for HC since the objective function is linear in δ_i's. Under this benign circumstance, how well will the GA and SA perform? For both algorithms, because we know that the maximum value is 1, we can stop it when the best solution gets close to 1 within a given tolerance.

• **GA** Since probability plays an important role in the GA, it might end up with quite a different solution if it starts with different initial seeds. To get an idea of the average performance of GA, we take a number of runs starting with different initial populations on a single rank-one matrix where $a_i^2 = \frac{1}{n}, i = 1, 2, \ldots, n$. The control tolerance is set at 0.01 and 20 runs are taken for each size. The results confirm that the variance in the initial population does cause a significant difference in the number of generations needed to get a good solution. However, even for the worst cases encountered in our tests, the growth in the computation cost approximates a straight line on the log-log scale against the problem size. The given tolerance was achieved within 100 generations for the problem of size 128.

We then run the GA on 20 random rank-one matrices for each size. The program is tested for tolerance 10%, 5%, and 1% and the results are shown in Figure 2.

Since a straight line on a log-log plot indicates a polynomial growth, we can say that for small and medium-sized problems

[3]This is approximately true in principle, but the B&B code we used takes longer. All programs are written in Matlab, and as an interpretive language, a more complicated algorithm is artificially slower. With suitable effort the B&B code could be written and compiled so that one branch cost approximately the same as one generation of the GA. This is an example of how the GAs simplicity favors it when programming effort is to be minimized.

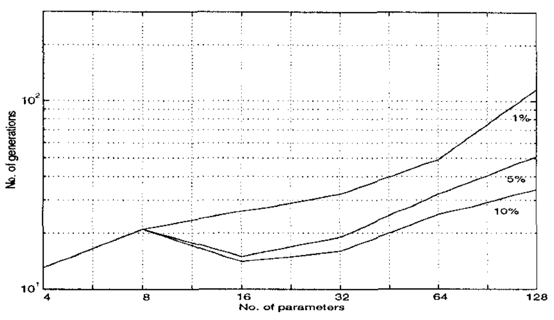

Figure 2: No. of generations versus problem size for various tolerances, for the worst problem out of 20 random rank-one matrices in each size. The vertical axis is on a $\log_{10}$ scale, while the horizontal axis is on a $\log_2$ scale.

($n \leq 128$) the average computation cost of GA for rank-one problems grows polynomially with the problem size. While this is much greater computation than B&B, this result encourages us to continue. The GA takes no special advantage of the rank-one nature of the problem, so we could optimistically hope that it might work similarly on the general case.

• **SA** For rank-one matrices, the smaller the initial temperature T_{max}, the less the steps the SA takes to get to the given tolerance. This is easy to understand since the objective function is linear now, the best thing to do is simply going up the hill at each step, which can be done by setting T_{max} small enough. For problems of size 128 and tolerance of 1% the average number of steps needed is less than 6, much better than the GA as expected.

4.2. General Random Matrices

Now we move our tests onto the random matrices we tested with the branch and bound technique so that comparisons between B&B and each of these three methods can be carried out. 10 random matrices are tested for each size of 8, 16, 32 and 64. Only the plots of size 32 are shown due to the limited space. But they are representative of other sizes. Let lb_{bnb} denote the lower bound we got from B&B while lb_{ga}, lb_{sa} and lb_{hc} being the best function value found by GA, SA and HC respectively.

• **B&B** The lower bounds usually start very close to their final values, e.g. mostly within 10% for size 32, which forms a big contrast to both GA and SA. And they normally can reach the final values after a few times of branching. The quick convergence of the lower bound is a feature of the B&B algorithm we use, though we can't be certain that it is indeed the global maximum. Most effort of the B&B is usually spent on bringing the upper bound down to make sure that the lower bound is near the global maximum. For $n = 32$, the normalized error between the upper bound and the lower bound gets down to within 5% after branching 74 times for the worst problem out of 50 we have tested. On average it takes about 16 times of branching to achieve this goal.

• **GA** For each matrix, the GA is run for up to 300 generations. Its performance degenerates significantly from the rank-one case. The best solution from the GA is usually lower than the lower bound from the B&B in the worst case, and it converges much more slowly. On average lb_{ga} approximates lb_{bnb} from below, and the number of generations needed increases with the problem size. Occasionally, the GA can find a solution slightly better than the B&B, but recall that the B&B was only run until it had reached a tolerance relative to the upper bound(1% for $n = 4, 8, 16$, 5% for $n = 32$, 10% for $n = 64$). There is no upper bound for GA, so it was simply run for a large number of generations.

Figure 3 illustrates the effect of the parameters p_c and p_m. It is the worst problem out of 10 matrices of size 32, with "worst" meaning the final ratio of lb_{ga} to lb_{bnb} is lowest even with the best parameter values found. Although some parameters perform better than the others in specific cases, no parameter choices stood out as the best across all matrices and problem sizes.

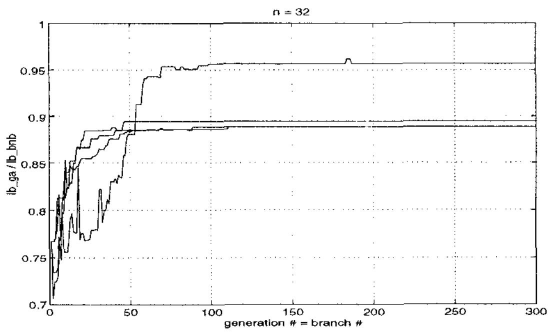

Figure 3: $\frac{lb_{ga}}{lb_{bnb}}$ versus no. of generations (= no. of branches) for the worst of 10 matrices of size 32. Different lines correspond to different sets of p_m and p_c and same initial population.

To see how the initial population affects the above performance we pick the above problem and repeat running the algorithm 10 times with the same parameters but starting with different seeds for random numbers. The result is shown in Figure 4. A good initial population does improve the performance of the algorithm in some cases, but the best lower bounds we get are still no greater than the lower bounds from branch and bound.

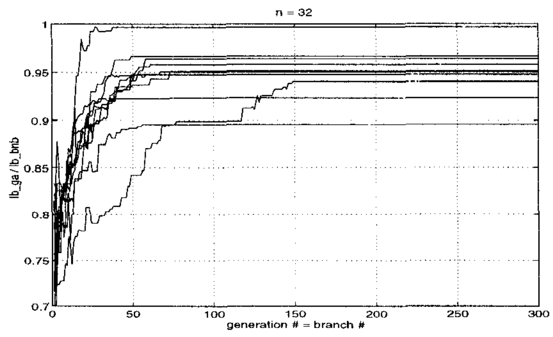

Figure 4: $\frac{lb_{ga}}{lb_{bnb}}$ versus no. of generations (= no. of branches) for the same problem as in Figure 3 but with 10 different initial populations.

• **SA** The simulation results of the SA present some similar features of the GA: the variance of the performance caused by the parameters and the random start, and the converging process. In the early steps, the SA seems to be more chaotic than the GA because of the relatively high temperature. For most of the problems the SA found the best solution that the B&B had found with the best initial guess out of 4. And for a few problems it found a slightly better solution than B&B did. But as we have explained the B&B was stopped when it reached a given tolerance relative to the upper bound while the SA was run until it converged. The problem shown in Figure 5 is the same as that in Figure 4. The SA simulation was carried out with 10 different initial guesses.

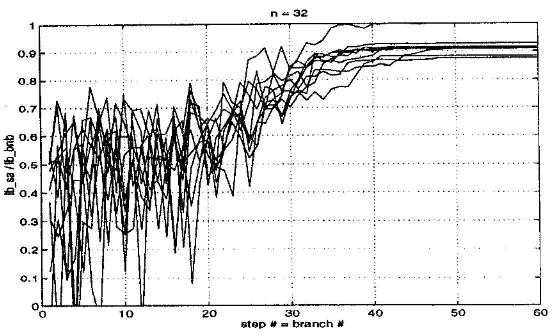

Figure 5: $\frac{lb_{sa}}{lb_{bnb}}$ versus no. of steps (= no. of branches) for the same problems as that in Figure 4 and with 10 different initial starting points.

• **HC** The hill climbing algorithm we are using is called *steepest ascent*, which starts at a random vertex and moves to the best of the n nearest neighboring vertices until a local maximum has been found. Since each step takes n function evaluations, it is equivalent to one generation in GA or one step in SA in terms of computation cost.

Figure 6 shows the results of HC applied to the same problem as shown above. Here 10 different random starting points were used for the HC. As expected, it gets stuck very quickly at local maxima. Although HC converges a lot faster, the average performance of GA and SA are much better in terms of finding the best solution.

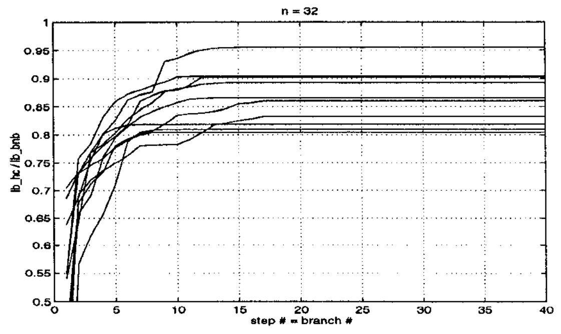

Figure 6: $\frac{lb_{hc}}{lb_{bnb}}$ versus no. of steps (= no. of branches) for the same problems as that in Figure 4 and with 10 different initial starting points.

5. Conclusion

We have explored the use of genetic algorithms and simulated annealing for a special case of robustness analysis. The main deficiency of both algorithms relative to branch and bound is their lack of an upper bound to give an indication of the quality of the lower bounds. Just considering the lower bounds, while both the GA and SA are not competitive with existing B&B methods, this study does support some of the claims for them. Specifically, they are easy to use and can give reasonably good answers for this particular NP hard problem. It does not support some of the hype about the quality of the solutions nor the efficiencies of the algorithms. This is illustrated in Figure 7, which is just a cartoon but is supported by the results in this paper. (The situation for SA is similar.) While B&B significantly outperforms these two stochastic approaches, it requires much greater programmer sophistication. Basically, GAs or SA could have been written, in principle, by a smart high school student, although they would likely have had no idea what the result meant. In contrast, understanding the problem and the B&B algorithm relies on theory usually taught at the graduate level.

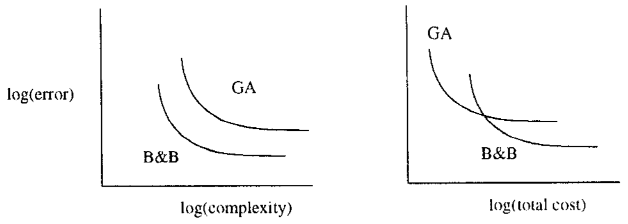

Figure 7: Cartoon of GA vs. B&B. Suppose we think of error as the deviation from the optimal. If we only take computation cost into account, the hard computing solution (B&B) beats the soft computing approach (GA). However, if we take programmers expertise into account, the picture changes and GA looks more favorable at the low end.

It would clearly be valuable to have more of a continuum between hard and soft computing, in that hard problems that are important are best treated using hard methods, while soft methods will be much more accessible to the vast majority of users. This complementarity should be exploited. One obstacle to doing this is that many advocates of soft methods deny the reality represented in the cartoon. They claim that as problems get more complex, only soft methods will work. Unfortunately, only those few who really understand *both* soft and hard methods realize how misleading this is, and they will have a very hard time explaining it to anyone who doesn't already know.

One positive result we hope to come out of further study along the directions of this paper is how some of the ideas from genetic algorithms and simulated annealing could be used to improve the B&B algorithms. In particular, the B&B algorithm in this paper is purely deterministic, and could potentially benefit from introducing some randomizing behavior of the stochastic approaches after optimizing to some extent the performance of the deterministic algorithm. While this probably won't make much difference on most problems, it would potentially make a large difference on occasional problems where the B&B algorithms performed poorly.

References

[1] Davis, Lawrence, (Ed.), 1987, *Genetic Algorithms and Simulated Annealing*, Pitman, London.

[2] Glodberg, David E., 1989, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison Wesley, Reading, MA.

[3] Mitchell, , 1995, "Genetic Algorithms: An Overview", *Complexity*, Vol. 1, No. 1, pp. 31-39.

[4] Newlin, Matthew P. and Young, Peter M., 1992, "Mixed μ Problems and Branch and Bound Techniques", *Proc. 31st IEEE Conference on Decision and Control*, pp. 3175-3180.

[5] Newlin, Matthew P. and Glavaski, Sonja, 1995, "Advances in the Computation of the μ Lower Bound", *Proc. American Control Conference*, pp. 442-446.

[6] Otten, R.H.J.M. and van Ginnekn, L.P.P.P., 1989, *The Annealing Algorithm*, Kluwer, Boston.

[7] Zhu, Xiaoyun, Huang, Yun, and Doyle, John, 1996, "Soft vs. Hard Bounds in Probabilistic Robustness Analysis", *Proc. 34th IEEE Conference on Decision and Control*, pp. 3412-3417.