# Genetic algorithms

***Simulating nature's methods of evolving the best design solution***

**Cezary Z. Janikow & Daniel St. Clair**

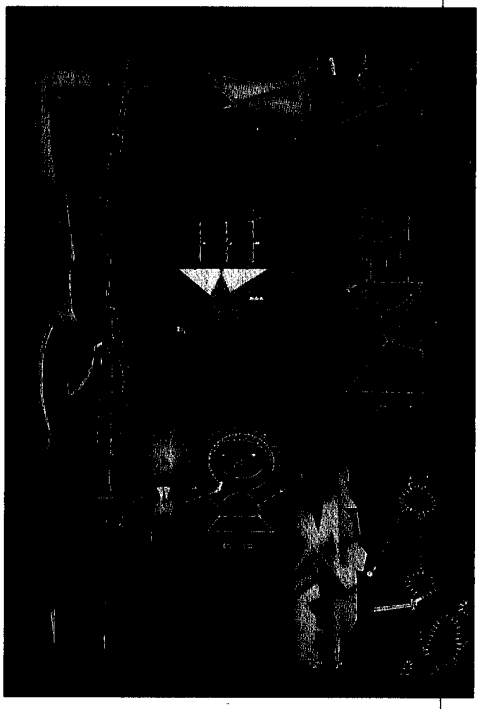
Gradually, problem solving is becoming dynamic agents interacting with the surrounding world rather than by isolated operations. Some methods are coming from nature, where organisms both cooperate and compete for environmental resources. This has led to the design of algorithms which simulate these natural processes. The genetic algorithm (GA) represents one of the most successful approaches.

Genetic algorithms are adaptive search methods that simulate natural processes such as: selection, information inheritance, random mutation, and population dynamics. At first, GAs were most applicable to numerical parameter optimizations due to an easy mapping from the problem to representation space. Today, they find more and more general applications thanks to: 1) understanding better the necessary properties of the required mapping, and 2) new ways to process problem constraints.

## GAs at a glance

A genetic algorithm operates as a simulation in which individual agents, organized in a population, compete for survival and cooperate to achieve a better adaptation. The agents are called *chromosomes*. The chromosome structure *(genotype)* is made up of genes. The meaning of a particular chromosome *(phenotype)* is defined externally by the user so that a complete chromosome represents a potential solution to a problem at hand. Traditional genetic algorithms operate on strings of bits.

Genetic algorithms use two mechanisms to provide for the adaptive behavior: selective pressure and information inheritance. Selection, or competition, is a stochastic process with survival chances of an agent proportional to its adaptation level. The adaptation is measured by evaluating the phenotype in the problem environment. This selection imposes a pressure promoting survival of better individuals, which subsequently produce offspring. Cooperation is achieved by merging information usually from two agents, with the hope of producing more adapted individuals (better solutions). This is accomplished by *crossover*. The merged information is inherited by the offspring. Additional *mutation* aims at introducing extra variability. Algorithms utilizing these mechanisms exhibit great robustness due to their ability to maintain an adaptive balance between efficiency and efficacy.



The Image Bank/Terry Allen

The simulation is achieved by iterating the basic steps of evaluation, selection, and reproduction, after some initial population is generated (see Fig. 1). The initial population is usually generated randomly, but some knowledge of the desired solution may be used to an advantage. The iterations continue until some resources are exhausted. For example, the simulation may be set for a specific time limit or a fixed number of iterations. Alternatively, if some information about the sought solution is available, the simulation may continue until some criteria are met. Finally, the population dynamics may be observed and the simulation may stop if convergence to a solution is detected.

A single iteration is illustrated in the bottom of Fig. 1, where the bullets represent individual chromosomes with intensity proportional to levels of adaptation—evaluation results. This evaluation is performed by a task-specific evaluation function. Each oval group represents the population instance at the single iteration. Stochastic selection (with replacement) is applied to the beginning population instance, producing the intermediate state. Because of the selective pressure favoring survival of better fitted individuals, the average fitness (manifested by darkness) of the chromosomes increases. However, no new individuals appear. Following the selection, reproduction operators are applied to members of the intermediate population. In this process, some chromosomes are modified. Therefore, the third population instance will finally contain some new chromosomes. This process continues for a number of iterations. The described iterative model is called the *generational* GA. Variations of this model are often used instead.

The two reproductive operators are visualized in Fig. 2, which assumes binary coding for a chromosome (white and black *genes)*. Mutation is performed here on the third bit of the


0278-6648/95/$4.00 © 1995 IEEE

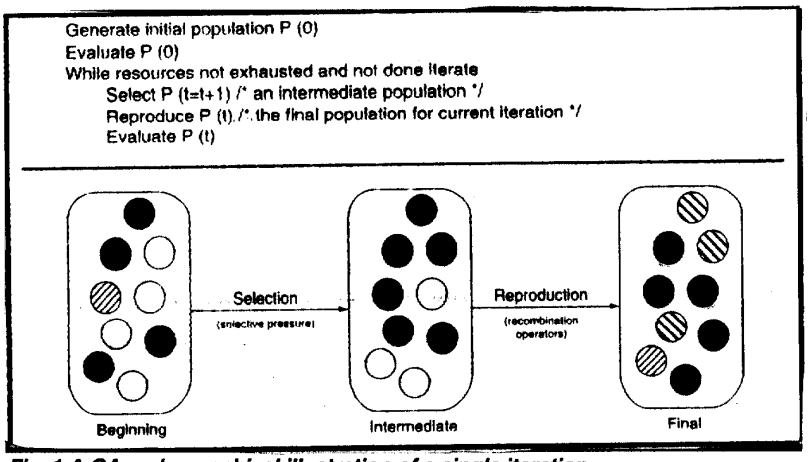



***Fig. 1 A GA and a graphical illustration of a single iteration.***

chromosome by flopping the *allele*. Crossover exchanges some genes between two chromosomes. Here, the exchange starts at the third bit. A mechanism is needed to apply the reproductive operators.

A simple approach is to use a stochastic firing mechanism with some prior probabilities for mutation and crossover. A more sophisticated approach is to update these probabilities based on history, or information contained in the population or individual chromosomes.

If these generic crossover and mutation operators are used, the only relation to the process at hand is the evaluation function providing the simulated environment. This is a great advantage, leading to domain-independent characteristics of the algorithm. This is also a great limitation, prohibiting use of available information about the problem.

## A theoretical and intuitive look

Genetic algorithms are not random searches. They explore regularities in the information the chromosomes represent. In a sense, the chromosomes are not really individuals but representatives of different species. Two different chromosomes may have similar adaptation levels if they represent similar species. However, the same two chromosomes may have different evaluations if the difference between them is significant. To explore such chromosome similarities, *schemata* are used. Schemata are similarity templates that contain fixed alleles for some genes but arbitrary alleles for others.

For example, Fig. 3 illustrates two different schemata in the top row—the shaded alleles represent the *don't care* positions. The left schema represents species that can only have two different chromosome instances, all shown below. The right schema is more general (actually, the left schema is a specialization, or subspecies, of the right one). A few of its representative-chromosomes are shown below. This schema can represent up to sixteen different chromosomes.

Unfortunately, schemata cannot be processed explicitly because they do not provide complete phenotype information needed for evaluations. Instead, a GA processes complete chromosomes. However, for practical problems, all possible chromosomes cannot be processed. Therefore, the information about individual chromosomes is generalized to draw conclusions about implicit schemata.

The selective pressure causes the search to proceed by working with increasingly representative chromosomes of the above-average schemata. The process continues by having more and more specific schemata represented in the population. For example, if all the chromosomes on the right of Fig. 3 evaluate high, a likely conclusion is that the third gene of the solution must contain a white allele and the fifth black.

The schemata can also be seen as hyperplanes of the search space. A schema with no fixed positions is a hyperplane that spans the complete search space. A schema with only one fixed position is a hyperplane that halves the search space, and so forth. For example, the right schema of that example, which has two fixed positions, represents exactly one-fourth of the possible number of chromosomes.

The iterative selection terminates when the represented schemata converge to a single most specific schema—a fixed chromosome. However, no schema can be reached that was not represented in the initial population. To extend the search to other schemata, the reproductive operators are used. Therefore, reproduction causes exploration of new schemata as well as generation of new instances of the present schemata.

Unfortunately, both mutation and crossover can disrupt currently represented schemata, in addition to generating new ones. Given a proper balance, the algorithm will continue exploring better and better hyperplanes. Because of the trade-off and the limited resources normally available for the search, there is no guarantee that the globally optimal chromosome will be found.

The hyperplanes identified during the search as those that are above-average provide *building blocks* (the fixed positions) for the algorithm. Then, the same iterative search can be seen as a process in which very short building blocks, those in the very general schemata, are put together to form longer and longer blocks (more specific schemata) until a particular chromosome is generated. This hypothesis is called the *Building Block Hypothesis*. Using building blocks, the reproductive crossover can be explained as a mechanism that assembles the building blocks identified by different chromosomes and promoted by selection.

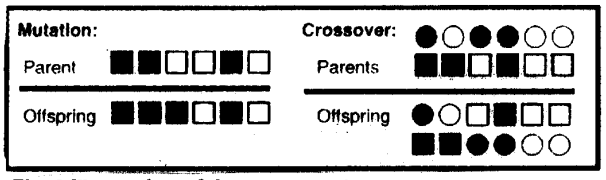



***Fig. 2 Illustration of the two operators.***

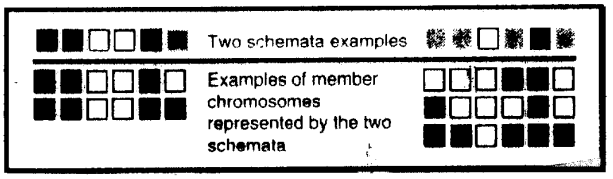



***Fig. 3 Illustration of the schemata concept.***

Since all this depends on the genes' locations in the chromosome, the crossover will minimize disruption of schemata with short building blocks (short substructures). Mutation introduces much smaller disruptions, especially for the more-general schemata. These properties guarantee that the above average schemata, which are promoted by the selection mechanism, will not be over-disrupted. This, along with the selection itself, explains why genetic algorithms work and is called the *Schemata Theorem.*

Because GAs work by processing implicit schemata by means of explicit chromosomes, and the number of schemata having chromosome representatives in a population of some fixed size is exponential, genetic algorithms are said to exhibit *implicit parallelism.* Genetic algorithms also exhibit *explicit parallelism,* that is, the processing can be parallelized. This property allows GAs to utilize fast-advancing parallel processing technologies.

## An example

The first example illustrates population dynamics as a search of the potential solution space. Consider a very simple problem of finding a maximum of an unknown function over integers 1 through 16, whose evaluation is the number of positive integers evenly dividing the argument—

f(n): [ 1,16]→[1,6]. For example, 4 can be divided by three different integers: 1, 2, 4. Thus, f(4)=3. The maximal value 6 belongs to the argument 12.

Furthermore, suppose we use a population of 10 chromosomes, each represented by a binary sequence of four bits $b_1b_2b_3b_4$. The value that the sequence codes is assumed to be the decimal equivalent of the binary number plus 1—giving the range 1 through 16. For example, 0010 represents the value 3. Fig. 4 plots the original function f(), along with chromosome-value distributions in the initial population, and after 10 and 25 generations.

This is a very simplistic example since the number of chromosomes is similar to the range of the function. Therefore, an exhaustive search could have been performed at a smaller cost. Nevertheless, it illustrates the dynamics of the population. Initially, the chromosomes are randomly distributed over the range, and the algorithm is said to perform *exploration* of the search space. After 10 iterations, the distribution concentrates around regions with higher expected payoff. After 25 iterations, all but a few chromosomes represent the sought maximum. These few solutions are results of random mutation. At this stage, the algorithm is said to perform exploitation of the search space. Actually, an important property of a genetic algorithms is that both exploration and exploitation are performed simultaneously during the search.

This example is actually an illustration of a problem not suitable for genetic algorithms. The search space of this problem was too small to justify the overhead of such a sophisticated algorithm. In fact, the GA performed worse than any exhaustive method would have. Problems suitable for GA applications are those with large search spaces, which cannot be searched exhaustively and for which no efficient algorithms exist.

## Applications

Genetic algorithms are applicable to problems that cannot be solved by other less expensive methods. The algorithm itself is a simulation which cannot provide real-time responses. Moreover, in general it can only find a "good" solution, that is an approximation of the solution. However, there is a wide variety of such problems for which any "close" solution is acceptable.

Genetic algorithms are most successful in numerical parameter optimization. The reason is that numerical solutions can be easily represented as linear chromosomes—both crossover and mutation act on linear sequences of alleles. Also, the quality assessment of such chromosomes is reduced to evaluations of the original function.

In general, a GA application requires:

1. A clear understanding of the problem and its objectives.
2. A genetic algorithm with
   a. chromosome representation with its semantics defined,
   b. evaluation function utilizing the representational semantics, and a selective pressure mechanism favoring better solutions,
   c. a population of randomly or otherwise generated representation structures—the chromosomes,
   d. reproductive operators, with some firing mechanisms often based on static probabilities.

A problem must be expressed in terms suitable for GA optimization. The most general problems suitable for applications are:

1. Search for a topological structure.
2. Numerical parameter optimization.
3. Combinations of these two.

Most problems can be mapped into parameter optimization problems. This is especially easy for those dealing with numbers. Moreover, such applications are the easiest to design, even in the domain-specific model, since only numbers are being processed.

However, a numerical problem does not necessarily imply a vector representation. Following the domain-specific model, the representation may be based on properties of the problem. A good example here is the transportation problem, where the objective is to set up transports between some sources and some destinations in such

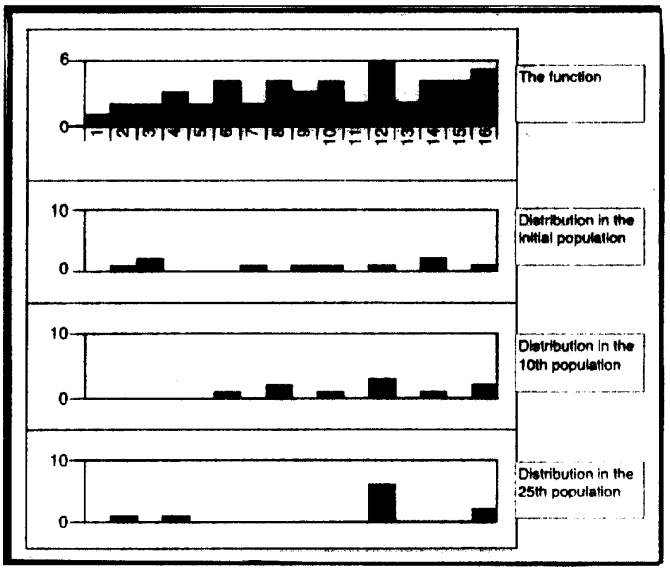



***Fig. 4 The original function and the chromosome distribution during simulation.***

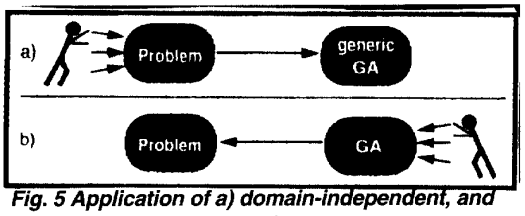



*Fig. 5 Application of a) domain-independent, and b) domain-specific GA.*

a way that the delivery costs are minimized and all demands are met. Chromosomes for this problem are best processed as two-dimensional arrays.

However, some problems are not easily mapped this way and must be processed as topological structures. This is the case, for example, for combinatorial problems such as the *travelling salesman* problem. Here the objective is to find a tour topology linking a number of cities in such a way that each city is visited exactly once and the tour's cost is minimized. Another example is the problem of learning inductive concept descriptions, where the objective is to find the best generalized description of provided examples. This case is much more difficult and requires some level of domain-specific processing since the operators need to be properly designed to process structures of the problem.

Yet, most practical problems are best represented as a combination of topological and numerical processing. An example is the problem of designing and tuning a neural network for feed-forward propagation. Here, the sought solution has to represent both the best network structure and its weights. Another good example is the problem of optimizing fuzzy rules for control or classification, where it is necessary to process high-level rules along with some numerical components such as rule weights.

## Level of independence

Most early GA applications were based on a *domain-independent* model. Dissatisfaction with its performance on real-life problems led to efforts to utilize some domain-specific information. This information may include rich heuristics, problem solving methodologies implemented in new operators, and a representation more suitable for the problem at hand.

One first successful application was work done on the travelling salesman problem. There, some reordering operators were used to modify the explicitly represented tour rather than blindly cutting and putting chromosomes together in hope of finding the right structure. The complexity of this approach is much greater, as it often requires designing a tailored representation, and a better understanding of the problem so specific methodologies can be implemented in the tailored operators.

On the bright side, this approach generally provides performance increases in orders of magnitude. For a number of problems, this is the only feasible approach. A good example of this approach is the machine learning system GIL, which has been designed for learning symbolic descriptions from examples (e.g., learning descriptions of patients with a specific disease from a hospital database). There, the chromosomes are symbolic concept descriptions and the operators implement inductive learning methodology. Figure 5 shows the two extreme approaches.

Another potential disadvantage of the domain-independent model is that the syntactical structures identified for the problem might not be quite suitable for applications of operators. In other words, even though this approach forces identification and processing of building blocks, such building blocks might be difficult to combine in some cases.

An example here is the problem of discovering the best topology of a neural network. In these cases an extreme effort must be made to designing a domain-specific model. This often causes the actual approach to settle somewhere in between these two extreme models. This trade-off is very often profitable in that the resulting algorithm exhibits satisfactory performance given the design effort.

## Constraints

In many applications, an additional difficulty is the constraints the sought solution must satisfy. Constraints cause the most serious problems in the domain-independent model, where the fixed representation is often incapable of representing only the desired chromosomes. This may cause the search to drift into improper regions of the search space.

The most common way to deal with these problems is to penalize chromosomes for not satisfying the constraints. This works quite well with *weak* constraints: those which can be violated to some extent. However, this approach often proves disastrous for *strong* constraints, which must be satisfied.

Another approach is to have follow-up routines for all the operators. This "fixes" the generated chromosomes by bringing them into the feasible space. However, for many applications, these routines are non-trivial and introduce additional computational complexity.

A better approach is possible in the domain-specific model. Here the representation is not fixed and can be bound to the problem. Thus, the genotype spans only the potential solution space. This action, alone, often causes the chromosomes to satisfy all the strong constraints. GIL uses this approach. In other applications, this may not be possible or sufficient. Then the obvious choice is to require the operators to be *closed* in the feasible space (i.e., produce only feasible offspring).

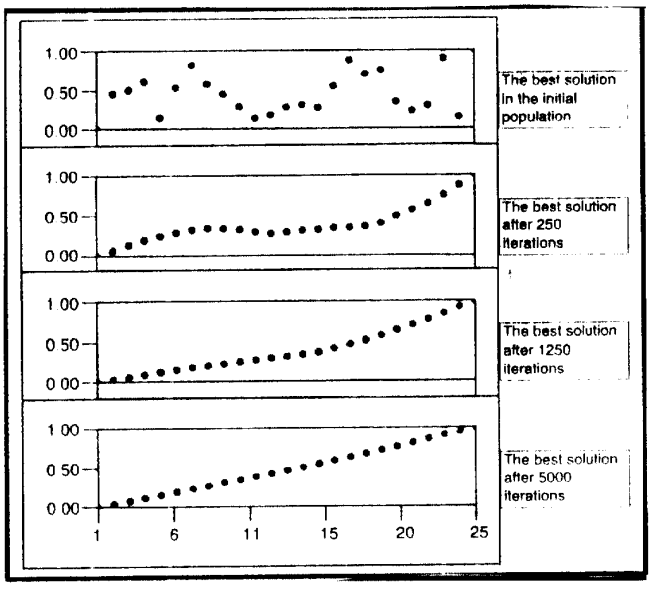



*Fig. 6 The best solutions found during the simulation.*

## GA variations

Today, there exist a number of variants of the generational approach to genetic algorithms. This is because many problems require special treatment. In the basic iterative simulation, two major alternatives emerged. First, the selective pressure is often based on ranks instead of actual values. This makes the algorithm's performance independent of some of the problem's unknown characteristics and allows control of the convergence speed. Second, the iteration may have only one operator, whose resulting chromosomes replace the weakest ones in the population (*steady-state* algorithm).

Other important modifications allow adaptation of the decisions that inherently affect GA performance: 1) probabilities of operator applications and their behavior, and 2) representation that promotes development and propagation of the building blocks.

Operator probabilities can be adapted by providing dynamic performance measures for the operators and linking their application to this measure. Adaptive operator behavior can be based on context detection. For example, GIL uses problem heuristics to trigger the most appropriate operators. Finally, adaptive representation can be provided by extending the representation, so that it becomes another parameter being optimized. (A good example is the so called *messy* GAs.)

An important GA extension is using new representation, which can be generic (e.g. the domain-independent model), but is more flexible than the originally proposed linear representation. The most studied and utilized is a tree-like representation borrowed from LISP programs. An additional advantage is that complete LISP programs do not require any additional drivers to utilize the generated information. This leads to hopes for automatic programming based on genetic algorithms.

## An application

As an application example, consider the problem of minimizing a function of $N$ variables with domains [0,1]. The function uses an index to measure the sum of squares of distances between all neighboring (by an index) variables. It can be shown analytically that the sum is minimized when all the distances are the same. To avoid the trivial case when all variables are the same, let us assume that the first variable must be zero and the last must be one (these are strong constraints). Then, the minimal sum will be produced when each variable $v_i$ has the value $v_i = v_{i-1} + 1/N$ (e.g., for $N$=5, the five variables have values 0, 0.25, 0.5, 0.75, 1). In other words, the optimal solution vector will span a straight line between 0 and 1.

To solve the problem, one must select a representation for a potential solution, provide operators to manipulate this representation, provide an objective function evaluating chromosome quality, provide a population of some initial chromosome-solutions, and iterate the algorithm.

Since a solution is a vector of $N$ bounded real variables, we decided on a floating point representation for a single variable, and a vector of such to represent a potential solution. To manipulate the representation, we provided: 1) a crossover operator with possible split locations between any two neighboring variables, 2) an operator averaging two vectors, and 3) a mutation operator modifying a single variable's value, within the domain [0,1], with non-uniform probability density. The modification's expected magnitude would decrease as the population aged.

As usual with numerical parameter optimization, the function itself provided the objective evaluations. Each chromosome was judged by its sum of squared distances. We decided on a population of fifty chromosomes, which were initialized randomly. To deal with the constrained variables, we narrowed the domains for the first and the last variable to [0,0] and [1,1], respectively.

A trace of a 5000-iteration simulation is illustrated in Fig. 6 for $N$=25, which presents values of the best chromosomes at some iteration intervals. The algorithm finds a plausible solution after a small number of iterations, then uses the remaining iterations to fine-tune this solution. In this constrained case of $N$=25, the absolute minimal sum is 0.041667. The best solution found after 5000 iterations was 0.041791.

## Summary

Genetic algorithms enjoy more and more successful applications, often in completely new fields such as machine learning and job scheduling. The most important factor in these advancements is building applications utilizing problem-specific representations, operators, and heuristics. At the same time, it is important to realize disadvantages and limitations of these approaches. The simulative nature prevents real-time applications and set-up costs do not stand up against other simpler approaches, if such are possible.

The experiments reported in this paper were performed using GenET—a genetic algorithm implementing different variants of the basic model. It also provides a library of representations and operators to choose from when writing an application. This system has been designed to speed up problem-specific genetic algorithm applications. An initial release, along with the user's manual, is available in public domain from the authors. Send inquiries to janikow@radom.umsl.edu with subject GenET.

## Read more about it


- Davis, L., (ed.), *Handbook of Genetic Algorithms.* Van Nostrand Reinhold, 1991.
- *Evolution Computation,* MIT Press, publishes articles on GAs and other evloution-based algorithms.
- Goldberg, D.E., *Genetic Algorithms in Search, Optimization & Machine Learning.* Addison Wesley, 1989.
- Grefenstette, J.J., "Genetic Algorithms and Their Applications," In A. Kent & J. Williams (eds.), *The Encyclopedia of Computer Science and Technology.* Marcel Dekker, 1990.
- Holland, J., *Adaptation in Natural and Artificial Systems.* University of Michigan Press, 1975.
- *Machine Learning* publishes a special issue on GAs every other year. Kluwer Academic Publishers. The 1993 issue describes the GIL system.
- *Proceedings of the International Conference on Genetic Algorithms,* which has been held every second year since 1985. Morgan Kaufmann.


## About the authors


Cezary Z. Janikow is Assistant Professor of Computer Science a the University of Missouri-St. Louis. His work focuses on genetic algorithms for numerical optimization with constraints and for symbolic concept learning.

Daniel St. Clair is Professor of Computer Science at the University of Missouri-Rolla. He also holds the position of Visiting Principle Scientist at McDonnell Douglas Research Laboratory in St. Louis.