GAT: A GENETIC ALGORITHMS TOOLKIT

Ying-Hong Liao *Chuen-Tsai Sun*

Department of Computer and Information Science,
National Chiao Tung University, Hsinchu, Taiwan 30050, R.O.C.
Email: liao@cindy.cis.nctu.edu.tw, ctsun@cis.nctu.edu.tw

ABSTRACT

During the last thirty years there has been a rapidly growing interest in a field called Genetic Algorithms (GAs). The field is at a stage of tremendous growth as evidenced by the increasing number of conferences, workshops, and papers concerning it, as well as the emergence of a central journal for the field. With their great robustness, GAs have proven to be a promising technique for many optimization, design, control, and machine learning applications. A genetic algorithms toolkit (GAT) has been developed to help researchers facilitate GAs. With the readily available tool users can reduce the mechanical programming aspect of simulation and concentrate on principles alone. The toolkit was established to help users operate and control not only the structural identification but also the parametric identification of GAs. It outlines how to design genetic algorithms and how to set parameters of different kinds of problems. The purpose of making this system available is to encourage the experimental use of genetic algorithms on realistic optimization problems, and thereby to identify the strengths and weaknesses of genetic algorithms. This paper describes the toolkit, shows how it can be used to solve various problems, and provides details on its implementation.

1 Introduction

Genetic Algorithms (GAs) are emerging as powerful tools to many real-world applications of diverse nature, and are increasingly being used in problem domains which can be defined in terms of search procedures and subsequent optimization of objective function(s) [1, 2]. With their great robustness, genetic algorithms have proven to be a promising technique for many optimization, design, control, and machine learning application.

Invented in 1975 by John Holland [3] to simulate biological evolution based on the concept of nature selection proposed by Darwin theory, GAs are now a rapidly expanding part of this general simulation trend. John Holland's simple GA only provides the abstraction and core of the GA. One faces numerous choices concerning how to proceed when one wants to apply the GA to a particular problem. Mitchell [4] wrote:

> "John Holland's simple GA inspired all subsequent GAs and provided the basis for theoretical analysis of GAs. For real problem solving and modeling, however, it is clear that the simple GA is limited in its power in several respects. Not all problems should use bit-string encoding, fitness-proportionate selection is not always the best method, and the simple genetic operators are not always the most effective or appropriate ones.

Herein, we consider developing a simulation tool, Genetic Algorithms Toolkit (GAT), to help researchers experiment as many experimental cases as possible in short time. This tool must be simple for the sake of easy operations. Moreover, the tool must be general enough to cover as many typical utilities as possible. By this tool researchers select system identifications (both structural identification and parametric identification) they want to simulate, and start the simulation. During the simulation the program returns information about the running status immediately and users learn in the reaction between they and computer. The tool presented in this paper has been developed to allow researchers to practice with a good simulator rapidly.

2 Genetic Algorithm Fundamentals

Genetic algorithms (GAs), which were inspired by biological evolution, are efficient domain independent search methods. That is, these methods could help us in effectively solving problem in different application domain. The goals of Holland's research [3] have been twofold: First, to abstract and rigorously explain the adaptive processes of nature systems. Second, to design artificial systems software that retains the important mechanisms of nature system. These methods are capable of applying in many fields and perform well. From the viewpoint of AI research, Holland's method provides a good mechanism of learning.

Figure 1 represents the GA evolution flow. GAs are population-based search techniques that maintain populations of potential solutions during searches. A string with a fixed bit-length usually represents a potential solution. In order to evaluate each potential solution, GAs need a payoff (or reward, objective) function that assigns scalar payoff to any particular solution. Once the representation scheme and evaluation function is determined, a GA can start searching. Initially, often at random, A GA creates a certain number, called the population size, of strings to form the first generation. Next, the payoff function is used to evaluate each solution in this first generation. Better solutions obtain higher payoffs. Then, on the basis of these evaluations, some genetic operations are employed to generate the next generation. The procedures of evaluation and generation are iteratively performed until the optimal solution(s) is (are) found or the time allotted for computation ends [5, 6].

The alteration process uses genetic operators to produce a new population of individuals (offspring) by manipulating the "genetic information," referred to as genes, possessed by members (parents) of the current population. It comprises two operations: crossover and mutation. *Crossover* recombinations a population's genetic material. The selection process associated with recombination assures that special genetic structures, called building blocks, are retained for future generations. The building blocks then represent the most fitted genetic structures in a population.

The recombination process alone cannot avoid the loss of promising building blocks in the presence of other genetic structures, which could lead to local minima. Also, it cannot explore search space sections not represented in the population's genetic structure. Here *mutation* comes into action. The mutation operator introduces new genetic structures in the population by modifying some of its genes, helping the search algorithm escape from local minima's traps. Since the modification is not related to any previous genetic structure of the population, it creates different structures representing other sections of the search space.

3 Major Procedures

This section describes details and implementations of important modules in GAT.

3.1 Representation

Usually, only two components of GA are problem dependent: the representation and evaluation functions. Representation is a key genetic algorithm issue because genetic algorithms directly manipulate coded representations of problems. In principle, any character set and coding scheme can be used.

```
procedure GA
begin
    t ← 0
    initialize P(t)
    evaluate P(t)
    while (not termination-condition) do
    begin
        t ← t + 1
        select P(t) from P(t − 1)
        alter P(t)
        evaluate P(t)
    end
end
```

Figure 1: Evolution flow of a genetic algorithm

GAT provides four gene encoding methods: binary coding representations, gray coding representations, real coding representations, and path coding representations [6, 7, 8].

3.2 Evaluation Function

Along with the representation scheme, the evaluation function is problem dependent. GAs are search techniques based on feedback received from their exploration of solutions. The judge of the GA's exploration is called an evaluation function. The notion of evaluation and fitness are sometimes used interchangeably. However, it is important to distinguish between the evaluation function and the fitness function. While evaluation functions provide a measure of an individual's performance, fitness functions provide a measure of an individual's reproduction opportunities. In fact, evaluation of an individual is independent of other individuals, while an individual's fitness is always dependent of other individuals.

To use GAT, the user must provide an evaluation function, which takes one gene array of a chromosome as input and returns a double precision value. There are two methods for user to specify the evaluation function: hand coded C/C++ function and mathematical formula. We will discuss how to write evaluation functions in GAT at section 4.

3.3 Initial Population

Choosing an appropriate population size for a genetic algorithm is a necessary but difficult task for all GA users. On the one hand, if the population size is too small, the genetic algorithm will converge too quickly to find the optimal solution. On the other hand, if the population size is too large, the computation cost may be prohibitive. The initial population for a genetic algorithm is usually chosen at random. In GAT user may choose initial population either generated by system randomly or provided by user.

3.4 Operators

From a mechanistic point of view, a GA is an iterative process in which each iteration has two steps, evaluation and generation. In the evaluation step, domain information is used to evaluate the quality of an individual. The generation step includes a selection phase and a recombination phase. In the selection phase, fitness is used to guide the reproduction of new candidates for following iterations. The fitness function maps an individual to a real number that is used to indicate how many offspring that individual is expected to breed. High-fitness individuals are given more emphasis in subsequent generations because they are selected more often. In the recombination phase, crossover and mutation perform mixing. Crossover reconstructs a pair of selected individuals to create two new offspring. Mutation is responsible for re-introduction inadvertently "lost" gene values. Most research has focussed on the three primary operators: *selection*, *crossover*, and *mutation*. While selection according to fitness is an exploitative resource, the crossover and mutation operators are exploratory resources. The GA combines the exploitation of past results with the exploration of new areas of the search space. The effectiveness of a GA depends on an appropriate mix of exploration and exploitation. The following describe these three operators in detail.

3.4.1 Selection

The selection phase plays an important role in driving the search towards better individuals and in maintaining a high genotypic diversity in the population. Grefenstette and Back [9] noted that the selection phase could be divided into the selection algorithm and the sampling algorithm. The selection algorithm assigns each individual x a real number, called the *target sampling rate,* $tsr(x,t)$, to indicate the expected number of offspring x will reproduce by time t. The sampling algorithm actually reproduces, based on the target-sampling rate, copies of individuals to form the intermediate population. In fact, there is some difference between an individual's actual sampling probability and its expected value. This difference is called bias. There are two types of selection algorithms: *explicit fitness remapping*, and *implicit fitness remapping* [10]. The first one re-maps the fitness onto a new scale, which is then used by the sampling algorithm. Proportional selection and fitness ranking belongs to this category. The second one fills the mating pool without passing through the intermediate step of remapping. Tournament selection belongs to this category. Tournaments are often held between pairs of individuals since using larger tournament has the effect of increasing selective pressure.

There are also several classification criteria, and selection strategies can be classified with respect to the following [11]:

- *Extinctive* versus *preservative* selection: The term preservative describes a selection scheme that guarantees a non-zero selection probability for each individual, i.e.; every individual has a chance of contributing offspring to the next generation. On the other hand, in an extinctive selection scheme some individuals are definitely not allowed to create any offspring, i.e., they have zero selection probabilities.

- *Left* versus *right* extinctive selection: In case of extinctive selection there is a major special case where the worst performing individuals have zero reproduction rates, i.e. they do not reproduce. This situation is referred to as right extinctive selection. Similarly, the best individuals are also prevented from reproducing in order to avoid premature convergence due to the existence of super-individuals. This is referred to as left extinctive selection.

- *Elitist* versus *pure*: A selection scheme that enforces a lifetime of just one generation on each individual regardless of its fitness is referred to as pure selection. In an elitist selection scheme some parents are allowed to undergo selection with their offspring [12].

GAT provides seven basic selection methods and five selection strategies. In GAT, a complete selection operator combines a basic selection and a selection strategy. The basic selection methods provided by GAT are: linear ranking selection, exponential ranking selection, proportional selection, sigma scaling selection, tournament selection, random selection and none selection. The four selection strategies provided by GAT are: left listist strategy, right elitist strategy, left extinctive strategy, right extinctive strategy, and none selection strategy.

3.4.2 Crossover

In order to explore other points in the search space, variation is introduced into the intermediate population by means of some idealized genetic recombination operators. The most important recombination operator is called crossover. A commonly used method, called one-point crossover, selects two individuals in the intermediate population which then exchange portions of their representation. Take on-point crossover as example. Assume that the individuals are represented as binary strings. In one-point crossover, a point, called the crossover point, is chosen at random and the segments to the right of this point are exchanged. For example, let x=101110 and y=0101100, and suppose that the crossover point is between bits 4 and 5 (where

the bits are numbered from left to right starting at 1). Then the children are x'=101000 and y'=0101010. Figure 2 illustrates this example.

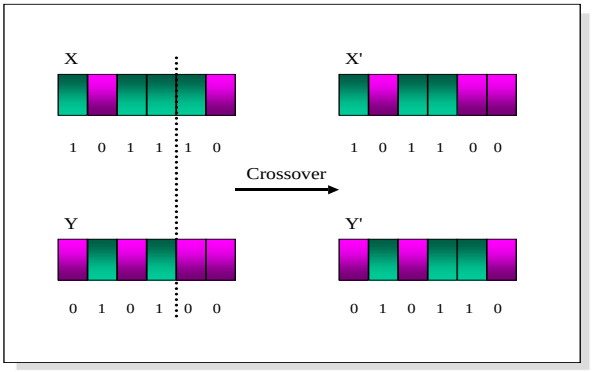

Figure 2: A one-point crossover example

Crossover serves two complementary search abilities. First, it provides new points for further testing upon hyperplanes already represented in the population. Second, crossover introduces representatives of new hyperplanes into the population. Crossover is, in effect, a method for sharing information between two successful individuals.

GAT provides thirdteen crossover operators: one-point crossover, two-point crossover, random-point crossover, uniform crossover, bitwise one-point crossover, bitwise two-point crossover, bitwise random-point crossover, bitwise uniform crossover, arithmetic crossover, partial-mapped crossover (PMX) [13], cycle crossover (CX) [14], order crossover (OX) [15], and none crossover. Where bitwise crossover means the crossover point may be any boundary of bit in a chromosome, while crossover point of plain crossover must be boundary of gene so that it can maintain the complement of genes in a chromosome. Also note that path coding GAs should use only PMX, CX, OX crossover methods to maintain the path representation property.

3.4.3 Mutation

When individuals are represented as bit strings, mutation consists of reversing a randomly chosen bit. For example, assume that the individuals are represented as binary strings. In bit complement, once a bit is selected to mutate that bit will be flipped to be the complement of the original bit value. For example, let x1=101010 and suppose that the mutational bit is bits 4 (where the bits are numbered from left to right starting at 1). Then the child is y1=101110. Figure 3 demonstrates this example. Another mutation called noising mutation applies only to naive encoding gene. This mutation adds a white noise to the gene selected for mutation.

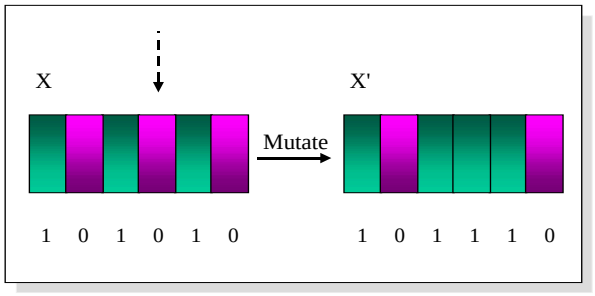

Figure 3: A bit-flip mutation example

If a number is represented by a group of bits in the string, small changes in the number's value are unlikely to follow from such mutation. This prevents the genetic algorithm from refining solutions to find an optimal solution after discovering good solutions in its neighborhood. The GA community usually views mutation as a background operator. Conversely, biologists view mutation as the main source of evolution.

GAT provides nine mutation methods: bit-flip mutation, random bit-flip mutation, non-uniform range mutation, uniform range mutation, inversion mutation [16], insertion mutation, displacement mutation, reciprocal exchange mutation, and none mutation. Note that path coding GAs should use only inversion, insertion, displacement, and reciprocal exchange mutation methods to maintain the path representation property.

3.5 Parameters

Running a genetic algorithm entails setting a number of parameter values. However, finding good settings that work well on one's problem is not a trivial task. There are two primary parameters concern the behavior of genetic algorithms: *Crossover Rate* (Cr) and *Mutation Rate* (Mr). The crossover rate controls the frequency with which the crossover the crossover operator is applied. If there are N individuals (population size=N) in each generation then in each generation $N * Cr$ individuals undergo crossover. The higher crossover rate, the more quickly new individuals are added to the population. If the crossover is too high, high-performance individuals are discarded faster than selection can produce improvements. However, a low crossover rate may stagnate the search due to loss of exploration power. Mutation is the operator that maintains diversity in the population. A genetic algorithm with a too high mutation rate will become a random search. After the selection phase, each bit position of each individual in the intermediate population undergoes a random change with a probability equal to the mutation rate Mr. Consequently, approximately $Mr \times N \times L$ mutations occur per generation, where L is the length of

the chromosome. A genetic algorithm with a too high mutation rate will become a random search.

GAT includes following parameters: optimization direction, population size, gene number in a chromosome, data types and formats of genes, crossover rate, mutation rate, simulation stop criteria, simulation logging. Detail about these parameters could be found in GAT manual.

4 Evaluation Function

To use GAT, the user must provide an evaluation function. There are two methods for user to specify the evaluation function: hand coded C/C++ function and mathematical formula.

4.1 Hand Coded Objective Function

User should providing objective function takes one gene array of a chromosome as input and returns a double precision value as fitness value of given chromosome. The object function *object()* must be coded in file <case>.cpp and the function prototype is:

```
double object(const Gene * const X, const
int n);
```

where "X" is gene array representation of the chromosome, "n" is the number of genes in "X". The body of the evaluation function is of course application dependent. Figure 4 shows a sample evaluation function that calculates the sum of gene value squares.

```
/*An example object() calculates the sum of
X[i]*X[i]*/
double object(const Gene * const X, const int n)
{
    register int i;
    register double Sum = 0.0f;

    for ( i = 0 ; i < n ; ++i )
    {
        Sum += X[i].get_value_double()
        X[i].get_value_double();
    }
    return (Sum);
}
```

Figure 4: An objective function for a sphere model

4.1.1 Mathematical Objective Formula

In addition to hand coded objective function, an evaluation function can be specified by simply designating a simple mathematical formula in the GAT. The mathematical formula must be written in a file named ⟨case⟩.exp, where ⟨case⟩ indicates the case name. The format of the objective function is:

```
[expression]
<expression>
```

where ⟨expression⟩ is the objective evaluation function. Figure 5 demonstrates how a sample evaluation function calculates the sum of gene value squares in a chromosome. The letter 'X' represents the chromosome while X[i] reveals the value of the ith gene of the current chromosome.

```
#An example formula calculates
#the sum of X[i]*X[i]
[exp]
sum( i, 1, 10, X[i]^2 )
```

Figure 5: An objective formula for a 10 dimension sphere model

5 Building GAT

GAT should run on most machines with an ANSI C++ compiler. This version has been complied and run successfully on UNIX/LINUX (using egcs1.1.x) and Win32 (using Borland C++ Builder and Microsoft Visual C++). This section will address about building the GAT systems. The code has been designed to be portable, but minor changes may be necessary for other systems.

5.1 Building for UNIX Systems

The GAT source archive includes a makefile that you can (and should) use to compile the GAT sources and build the "gat" and "casegen" program files. This makefile invokes the ANSI C++ compiler (g++) to produce object code files. If you get warnings or error messages, this is usually a bad sign. Some compilers issue warnings just because you ask for ANSI compilation. If you get any other error messages, please let us know.

5.2 Installation for Win32

You may run "make -fmakefile.bcb" to build the GAT by Borland C++ Builder compiler, or run "nmake -fmakefile.vc" to build the GAT by Microsoft Visual C++ compiler.

6 Running the programs

The GAT flowchart presented in Figure 6 demonstrates the GAT reads two input files to create the simulation and produces two output files containing the processing details of the simulated GAs. Details concerning the interface between the user-written function and the GAT are explained below.

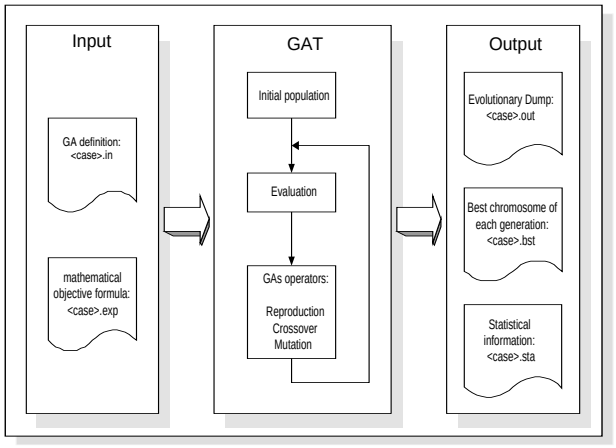

Figure 6: Flowchart of the Genetic Algorithms Toolkit.

Before running the GAT, execute the "casegen" program, which prompts you for a number of input parameters. All of this information is stored in a file for future use, so you may only need to run "caegen" once. A <Enter> response to any prompt gets the default value shown in brackets. The prompts are "the case name ["default"]:". If a string is entered, say "foo", then the files for this case will have names like "foo.in", "foo.in", "foo.bst", etc. Otherwise, the file names are "default.in", "default.in", "default.bst", etc.

6.1 Options

GAT allows a number of options, which control the kinds of output produced, as well as certain strategies employed during the search. Each option is associated with a single character. Responding to the "options" prompt with a string containing the appropriate characters indicates the options. If no options are desired, respond to the prompt by typing <Enter> key. Options may be indicated in any order. All options may be invoked independently, except as noted below.

- "a": evaluate all structures in each generation. This may be useful when evaluating a noisy function, since it allows the GA to sample a given structure several times. If this option is not selected then structures, which are identical to parents, are not evaluated.
- "b": write the best and average value to the standard output after each generation.
- "c": dump cooked bit string to <case>.out file.
- "d": dump the last generation to <case>.ini file.
- "e": calculate objective expression from file <case>.exp. This allows the user to restart the experiment at a later time, using option "i".
- "i": read chromosomes into the initial population. The initial population will be read from the <case>.ini file. If the file contains fewer chromosomes than the population needs, the remaining chromosomes will be initialized randomly. Note: it is good practice to allow at least some randomness in the initial population.
- "r": dump raw bit string to <case>.out file.

6.2 Input and Output Files

- <case>.in - contains input parameters. This file is required.
- <case>.out - intermediate checkpoint files produced when the number of saved dumps is greater than 1 and the dump interval is positive. <case>.ini is always identical to the latest dump if option "d" is set. The number of chromosomes in <case>.out is indicated by the response to the "number of dumping chromosomes in each generation" prompt during casegen. This file is produced if the number of saved chromosome is not zero.
- <case>.exp - contains the mathematical formula of the objective function. This file is read and used only when the option "e" is set.
- <case>.ini - contains chromosomes to be included in the initial population. This is useful if you have heuristics for selecting plausible starting chromosomes. This file is read only when the option "i" is set. This file can also be produced as a snapshot of the latest population, if the "d" option is set.
- <case>.bst - contains the best chromosome in each generation found by the GAT.
- <case>.sta- contains data describing the performance of the GAT. The <case>.sta file contains the best fitness, the average fitness, the worst fitness, and standard deviation of each population.

6.3 Running GAT under UNIX

The UNIX makefile produces a random archive called libgat.a that can be linked to the user's evaluation function. Suppose the new evaluation function is in file case1.cpp. Then the command "make CASE=case1" will create an executable called "gat.case1".

To make things easier, a shell script called "go" is provided. This command takes two arguments: the first is the root name of the source file containing the evaluation function; the second is an optional file suffix (as discussed under setup above). The command "go case1" will compile the user's evaluation function, link it with the GAT random archive, run the program using the case1.in input files, and produce reports in files

case1.bst, case1,env, and case1.sta. The reporting files are discussed below.

6.4 Running GAT under Win32

The Win32 version of GAT assumes a more rudimentary make facility. Suppose the new evaluation function is in file case1.cpp. Then the following command will create an executable called "gat.exe": "make -fmakefile.bcb -DCASE=case1". Once the executable file is made we can execute the program by running GAT.EXE with an optional command line argument: "GAT.EXE <case>". This approach can also be used to change other functions in GAT. There is also a batch file GO.BAT provided that will compile and run the case as "go" in UNIX environment.

7 Conclusions and Ongoing Work

We have provided a general-purpose GAs toolkit, which serves as a generic utility for applications involving optimization. The toolkit can be used for both - fast development of prototypes for experimentation as well for developing applications. The toolkit design supports the state of the art of genetic optimization techniques. The proposed model can also be adopted as a problem-solving scheme.

We are currently investigating two enhancements to the GAT: adding options to the tool so it will cover a wider class of problem domains in a more flexible manner and porting this tool to a platform independent program suitable for interaction via the standard World Wide Web interface.

8 Acknowledgment

Research project funded by National Science Council, Taiwan, NSC 89-2520-S-009-006.

References

[1] S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, pp. 671–680, 1983.

[2] J. J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities," in *National Academy of Sciences*, 1982, pp. 2554–2558.

[3] John H. Holland, *Adaptation in Natural and Artificial Systems*, The University of Michigan Press, 1975.

[4] Melanie Mitchell, *An introduction to genetic algorithms*, MIT press, 1996.

[5] J. J. Grefenstette, "Optimization of control parameters for genetic algorithms," *Transactions on System, Man, and Cybernetics*, vol. 16, no. 1, pp. 122–128, 1986.

[6] David E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Reading, Massachusetts. Addision-Wesley, 1989.

[7] H.J. Antonisse, "A new interpretation of schema notation that overturns the binary encoding constraint," in *Proceedings of the Third International Conference on Genetic Algorithms*, J. Schaffer, Ed., San Mateo, CA, 1989, Morgan Kaufmann Publishers.

[8] Zbigniew Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs*, Springer, third, revised and extended edition, 1996.

[9] J. J. Grefenstette and J. E. Baker, "How genetic algorithms work: A critical look at implicit parallelism," in *Proceedings of the Third International Conference on Genetic Algorithms and Their Applications*, J. D. Schaffer, Ed., 1989, pp. 20–27.

[10] D. R. Bull D. Beasley and R. Martin, "An overview of genetic algorithms: Part 1, fundamentals," *University Computing*, vol. 15, no. 2, pp. 58–69, 1993.

[11] T. Back, F. Hoffmeister, and H. P. Schwefel, "A survey of evolution strategies," in *Proceedings of the Fourth International Conference on Genetic Algorithms and Their Applications*, R. K. Belew and L. B. Booker, Eds., july 1991.

[12] K. A. De Jong, *An Analysis of the Behavior of a Class of Genetic Adaptive Systems*, Ph.D. thesis, University of Michigan, 1975.

[13] D.E. Goldberg and R. Lingle, "Alleles, loci, and the tsp," in *Proceedings of the First International Conference on Genetic Algorithms*, J.J. Grefenstette, Ed., 1985, pp. 154–159.

[14] I. M. Oliver, D. J. Smith, and J.R.C. Holland, "A study of permutation crossover operators on the traveling salesman problem," in *Proceedings of the Second International Conference on Genetic Algorithms*, J.J. Grefenstette, Ed., 1987, pp. 224–230.

[15] L. Davis, "Applying adaptive algorithms to epistatic domains," in *Proceedings of the International Joint conference on Artificial Intelligence*, 1985, pp. 162–164.

[16] M. Herdy, "Application of the evolution strategy to discrete optimization problems," in *Proceedings of the First International Conference on Parallel Problem Solving from NAture (PPSN)*, H.-P. Schwefel and R. Manner, Eds., 1991, vol. 496.