1993

Genetic Algorithms and the Satisfiability of Large-Scale Boolean Expressions.

Thomas A. Bitterman
Louisiana State University and Agricultural & Mechanical College

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

U·M·I

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

Order Number 9405386

Genetic algorithms and the satisfiability of large-scale boolean expressions

Bitterman, Thomas A., Ph.D.

The Louisiana State University and Agricultural and Mechanical Col., 1993

U·M·I

300 N. Zeeb Rd.
Ann Arbor, MI 48106

GENETIC ALGORITHMS AND THE SATISFIABILITY OF LARGE-SCALE BOOLEAN EXPRESSIONS

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science

by

Thomas Bitterman
B.S. Kent State University 1988
August 1993

Acknowledgments

Writing a dissertation is an exhausting process for not only the writer but also for all around him. The author would like to thank his advisor Sukhamay Kundu for his patience and time. Thanks are also due to my parents for their moral and financial support without which this would have been impossible. Thanks are also due to Molly Molloy for a great deal of patience and understanding during the preparation of this manuscript and for invaluable advice on research methods and information sources. I was also like to thank Michael Land for his generous instruction in the art of ju-jutsu, which has enabled me to gain the rather low level of expertise I currently have. Last, but not least, I would like to thank all the friends I have made during my graduate career. Even though I came from "up North" they welcomed me as a friend and helped keep me sane during this ordeal by lending a needed sense of perspective.

Contents

Abstract

The two new genetic methods (i) overpopulation and (ii) bitwise expected value are introduced. In overpopulation, we create a temporary population of size Mn (M>1) via genetic operators and we select the n children with the highest estimated fitness values as the next generation. The rest are discarded. Bitwise expected value (bev) is the fitness estimation function used. Overpopulation and bitwise expected value are applied to the NP-complete problem 3SAT (a special form of Satisfiability in which the boolean expression consists of the conjunction of an arbitrary number of clauses where each clause consists of the disjunction of 3 boolean variables) with excellent empirical results when compared to the performance of the standard genetic algorithm. Overpopulation increases the cost of producing each generation due to the overhead required to maintain the larger temporary population, but results in many fewer generations to solution. Using bitwise expected value as a fitness estimator causes the algorithm to take slightly more genrations to solution but is much faster to calculate than the fitness function, leading to a decrease in wall-clock time to solution. Theoretical justification for the success of overpopulation is seen as a result of a generalization of the schema growth equation. Bitwise expected value is seen as an analogy to the Building Block Hypothesis. Empirical evidence of high correlation between bev and the fitness function is presented. We also introduce the target problem concept, in which a difficult problem is transformed into a well-known problem for which a good genetic method of solution is known. As an example of the target problem concept a transformation from the Traveling Salesman Problem to Satisfiability is demonstrated. Overpopulation and bitwise expected value are applied to the resulting boolean expression, with good results. An interesting convergence property is observed.

1. Introduction

When computers were first envisioned they were a means to solve mathematical problems which required too many operations for even a team of people. The emphasis was on the speed which the computer could bring to such problems as calculating logarithms and statistical analysis. The performance of a computer on such problems is many orders of magnitude greater than any possible human effort. The size of the problems computers were asked to solve quickly grew beyond the ability of hardware designers to make faster computers. In particular, a field sprang up whose purpose was to make computers do things that no one else had ever thought a computer could do, things that seemed to require intelligence when performed by people. This field is Artificial Intelligence (AI), and it has stretched computing power to the breaking point.

The methods devised to cope with problems in AI share many features with the problem solving methods in other fields. There are statistical categorization procedures, vision algorithms that depend upon the physics of light, non-monotonic theorem proving systems, and a large number of programs that play games. Almost every field which involves problem solving has had some impact on AI, whether it be as a model for a new search technique or the problem domain of an AI program.

The most important feature of traditional AI programs is their almost continuous searching ([37, 39]). AI has been called "the science of searching" because so many of its techniques involve searching for the optimal solution among many candidates. For example, when a computer vision program is faced with a scene, it searches for an interpretation of that scene which fits best with its built-in knowledge of the types of elements it might encounter. It is given a set of lines and shadings, and it uses its knowledge of the way objects look in a given light to generate possible explanations of the scene, which it then evaluates and uses as a basis to for action in the world. As the robot moves the shading changes and so a new search has to begin for an explanation of the new shading pattern.

Traditionally, search algorithms have conceptualized the search space as a tree with each leaf representing a candidate solution, each interior node an intermediate step in finding a solution, and the branches as allowed steps in the search. This is an analogy to formal systems in logic and mathematics, where you are given a set of axioms (corresponding to the known facts about the world), a set of inference rules (corresponding to the allowed steps in the search) from which is derived a set of intermediate results (corresponding to interior nodes in the search tree) and one or more final conclusions as a goal (corresponding to leaf nodes on the search tree). The two major strategies are depth first search and breadth first search. Depth first search follows one path down the tree until it comes to a dead end and then backs up to the first untried branch, starting a new depth first search from that point. Breadth first search tries all candidate solutions reached after n steps before it tries any of the candidate solutions reached after n+1 steps. These strategies have had many notable successes. They play excellent chess and a form of these strategies, backward chaining, is at the heart of many expert systems.

As the problems posed to computers have become more complex and the fields in which computer power is being used have multiplied, the domain of search has grown. The earliest programs dealt with relatively small "toy" domains, with a few blocks and a simulated robot arm. Today's programs are called upon to deal with air traffic control, the fluctuating stock market, and a possible mission to Mars. The search for near-optimal answers becomes more and more difficult as the domains get larger, and more stands to be lost on a wrong answer.

The increasing size of the domains encountered revealed the flaw in depth-first and breadth-first search: as the domain grows larger the solution space grows exponentially. The traditional algorithms could guarantee optimal answers for small domains but were incapable of being scaled-up to larger problems.

The traditional answer to the increasing pressure put upon search algorithms by larger domains has been to want faster computers. "If only we had faster processors," became the lament of AI researchers faced with a larger problem. This steadily

squeezed researchers out of the field, because it was felt that any sort of important work needed to be done on big machines. This threatened to leave AI an elitist field only entered into by those few with the requisite budget.

An alternative answer is the parallel approach. Parallel algorithms cannot avoid the exponential growth of the problem domain (no known algorithm can, and it is possible that no algorithm at all can), but they can sub-divide the problem into more reasonably sized chunks. Each processor works on its own little chunk of the problem, and the sub-answers thus obtained are combined to form an acceptable (perhaps sub-optimal) answer. At first, parallel computing was even more expensive than serial computing. No one knew exactly how to write efficient parallel code, or what architecture was best suited for different problems, or how to generate good parallel algorithms from known serial ones. The first efforts at generating parallel search algorithms were straightforward attempts at rewriting serial code to take advantage of a multi-processor system. It soon became apparent that this was not opening up any new ways of thinking about the big problem: what is a good way to search?

As computing power has become more and more affordable, there has been increasing interest in the use of non-traditional parallel algorithms such as neural nets ([42, 43]). These new algorithms often draw their source of inspiration from the natural world in which many things are happening at once, as opposed to the symbolic world of mathematics and logic. Work on these new algorithms is still in its infancy, and many basic theoretical results are lacking.

1.1 Why Genetic Algorithms?

One of the most promising of these parallel algorithms is the genetic algorithm (GA) ([1-6, 38, 40, 41, 44]). Based upon analogy with biological evolution, GAs are a robust search/learning algorithm that has been shown to exhibit good performance in a wide variety of domains. Nature started out with simple atoms and has formed molecules, polymers, amino acids, uni- and multi- cellular organisms, plants, insects, and fi-

nally humanity. That nature could start from such humble origins to create such complex organisms is certainly a sign that natural processes are capable of great things. We could do worse than to imitate nature.

GAs show interesting properties of emergence, that is, they show the ability to display behavior that is adaptive in a situation unforeseen by the programmer, or even the ability to discover new solutions to problems. The best known example of emergence in the physical world is the ideal gas laws. Individual gas molecules have neither temperature, pressure nor volume, but a collection of them in a container does. In the same way, one organism does make not an ecosystem. Nor will one artificial organism make an artificial ecosystem, but a collection of them left to evolve may surprise the programmer by exhibiting behavior which was not explicitly programmed into the system.

GAs are efficient problem solvers, with a built-in feature known as *implicit parallelism* (described in more detail in section 2.2). Search tends to be directed toward several different promising areas of the search space in parallel, even though little knowledge of the problem is contained in the algorithm.

GAs are a very general method of optimization, good over many different applications with little adjustment needed. These are good features when the problem at hand is large, and all the variables affecting the decision cannot be known in advance. Also, any discovery made about the properties of genetic algorithms will apply to any field in which a GA is used. This is not often true for more specialized optimization techniques, about which much may be known, but whose area of applicability is so small as to make generalization difficult and of limited value in any event.

GAs are adaptive, that is, they can accept feedback from the environment and change their behavior in an acceptable manner. Systems in which a great deal of domain knowledge is encoded in rigid data structures often show great ability in certain limited areas, but fail to go beyond that area because they are unable to adapt to the new conditions that new environments contain. This problem occurs because the system tries to pigeon-hole everything it encounters into pre-determined categories. What

is needed is a system which can not only learn, but which can change its knowledge representation in order to fit changing conditions. In a GA, only the best solutions at each generation live to pass on genetic material to the next generation. Solutions that no longer are relevant to the current situation will be discarded to make way for more timely ones. This includes solutions to the question of: how should the system organize its knowledge so as to maximize its overall fitness?

GAs are robust because they spread knowledge of the environment over many different genes, thus minimizing the disruption caused by unexpected occurrences. Noisy data has limited effect upon the algorithm because the fitness maximization occurs over many genes, thus allowing fluctuations in input values to even out over many samplings. Algorithms which are specially designed in order to take advantage of certain properties of the problem space can often be fooled when feedback is imperfect. This is a particularly important problem in speech recognition and comprehension where the signal to noise ratio can be very low. A genetic algorithm can adapt to noisy speech input by gradually choosing those genes that do the best job of recognizing phonemes and building upon those genes to obtain new genes that recognize suffixes and prefixes, continuously building upon good solutions to small problems to form good solutions to bigger problems.

In this dissertation the author will introduce two new genetic methods, bitwise expected value and overpopulation, and apply them to an optimization problem known to be NP-complete, 3SAT. The author believes that many of the representation issues that concern so much of current genetic algorithm research can be obviated by the transformation of the original problem into an isomorphic problem for which a good representation can be easily devised. As an example of such a procedure, a transformation from the Traveling Salesman Problem (TSP) to SAT will be described, and Bitwise Expected Value and Overpopulation will be applied to the resulting boolean formula.

2. Genetic Algorithms

Genetic Algorithms (GAs) are search algorithms based upon an analogy with biological mechanisms of natural selection. We loosely define a species as a group of organisms, each member of a species having the ability to mate with any other member of the same species to produce viable offspring. A population is a subgroup of a species S consisting of those members of S which actually interact. S may consist of more than one population because distance or geographical barriers prohibit some subgroups from interacting.

In nature organisms live in an environment and compete both with the environment, to survive, and with other members of the population, to secure access to limited natural resources such as food, water, shelter, and available mates. Success on an individual level is defined as the number of offspring an individual contributes genetic material to. Success depends upon the performance of the individual in the environment it is faced with relative to the performance of other individuals in the same population. For example, given a polar environment, organisms that are better adapted to cold weather (having thicker coats or higher body fat) will tend to be better fed and more active, thus leading to the acquisition of more mates and the production of more offspring relative to other members of the population lacking those traits. These desirable (for a polar environment) traits will be passed on through the genetic material contributed by successful individuals to their offspring. This leads to a gradual increase in the percentage of individuals exhibiting desirable traits. This simple example illustrates why the proportion of individuals with a desirable trait in a population increases from generation to generation under natural selection.

The *population* of a genetic algorithm consists of a set of genes. Each gene is of the form $a_1a_2...a_n$, where each a_i comes from an alphabet A and is called an *allele*. We write γ for a gene and $\Gamma=\{\gamma_1, \gamma_2, ..., \gamma_n\}$ for a population. The *representation* maps genes onto trial solutions to a problem. As the representation is usually constant we

will use the terms *gene*, *string*, and *individual* interchangeably to refer to both the actual gene and the trial solution it represents. The *fitness function* is the method used to determine the goodness of each trial solution as a solution to the problem at hand. When we speak of the fitness of a gene γ we are referring to the fitness of the trial solution that γ represents. The fitness function is designed to give the best fitness scores to the individuals that are the fittest in the given environment, that is, individuals that represent good solutions to the problem to be solved are given higher fitness scores than individuals that represent poor solutions. Mating is accomplished through one or more methods which produce new genes from old genes. These methods are called *genetic operators*. The *environment* of the GA is the problem to be solved.

The first step in using a genetic algorithm is to decide on a suitable representation of the trial solutions in terms of a string of alleles. A good representation is one that is clear and computationally inexpensive. Designing a good representation is, in large measure, still an art.

Once a representation has been decided upon it is necessary to pick the genetic operators. Genetic operators are called upon at the end of every generation in order to form new genes from the genes present in the previous generation and it is important that the operators act in concert with the representation. If the operator(s) the designer has chosen create unfit children from fit parents, or fail to produce children who are more fit than their parents, he has failed at the operator selection task. When a genetic operator is applied to one or more genes $\gamma_1, \gamma_2, ..., \gamma_n$ we say that $\gamma_1, \gamma_2, ..., \gamma_n$ are *mating*.

A1: The Standard Genetic Algorithm (SGA)

1: Generate a (possibly random) initial population P.

2: While no gene in P is a satisfactory solution do

begin

create the next generation P′ from P;

$P \leftarrow P'$;

end;

The standard genetic algorithm contains two phases. The first phase consists of the production of an initial population. The most important characteristic the initial population must have in order for the algorithm to exhibit good performance is variety. In particular, for every value x in A and for every allele a_i, a_i should have value x in some gene γ_j in the initial population. Genetic algorithms capitalize on the difference in fitness values between genes within a population so it is important to have a heterogeneous starting population. Often the initial population is created at random as GAs are almost totally insensitive to initial conditions.

The second phase is called a *generation*. The generation phase is repeated until a satisfactory solution is found.

In an random initial population, each allele a_i in each gene takes on a value $b \in A$ with probability $\frac{1}{|A|}$. The research to be described uses the binary alphabet A={0, 1}. One of the major advantages to a random initial population is the high probability that every member of A will be found in allele a_i of some gene in the population. If the desired solution required value 1 in allele a_i, and all initial genes had value 0 for allele a_i, then it would be very difficult for the algorithm to reach the desired solution under most genetic operators. In addition, a random start creates a lot of variety in the initial population. The selection process works upon the differences in gene fitness so a greater range of fitness values will make the selection process more efficient at finding better solutions. Lack of genetic variety occurs in natural populations also, and the mating practice that results from a lack of genetic variety in parents is known as *inbreeding*.

Creation of the next generation P′ from P requires 5 steps, as shown below.

A2: The creation of the next generation

1: Apply the fitness function to each individual in P.

2: Assign each individual in P a fitness score, which may be different from the fitness function value.

3: Select individuals from P for reproduction according to fitness score.

4: Apply genetic operators to the selected individuals to obtain new individuals.

5: Replace (part or all of) P with the new individuals.

Competition enters the SGA through the fitness function. There are only a limited number of genes that will be chosen to reproduce. The fitness function evaluates the genes, assigning each a fitness value. Genes are chosen for mating in proportion to their fitness, so a high fitness value raises the probability that a gene will mate. In the algorithm used in the research to be described each individual is evaluated independently of competing individuals. To have a fitness score each gene must first be interpreted by using the representation to map it onto a candidate solution. The fitness function then evaluates the goodness of that candidate solution as a solution to the problem at hand. All that is needed for a satisfactory fitness function is a way to rank the candidate solutions in the order in which they are satisfactory solutions to the problem at hand. An example of a maximization problem is the function $f(x)=x^2$, which is to be maximized. The representation would use alleles from {0, 1} and map each gene onto an integer, interpreting each γ_k as the base 2 coding for some integer i. The fitness function would determine the fitness of γ by squaring i. For example, the individual 0110 would represent the number 6 and have fitness f(x)=36.

Once the individual is evaluated by the fitness function, the fitness score is determined by scaling the output of the fitness function in some way. The purpose of this scaling is to ensure that the population is not overwhelmed by an extraordinary individual in the first few generations. In a random initial population there is always the chance that one individual will be significantly more fit than the rest. This individual will produce proportionately more copies of itself as it will be picked proportionately more often to mate. Unfortunately, soon the population will consist of nothing but near-copies of this successful individual. This will effectively end the search because there

will not be adequate genetic diversity for the genetic algorithm to exploit. This successful individual is not necessarily the optimal solution, or even an acceptable one. This phenomena, where a single individual comes to comprise most or all of a population in the early stages of search, is known as *premature convergence*. To prevent premature convergence fitness values can be scaled so that the ratio of the most fit to the least fit individual is constant over generations. Scaling fitness values is a bit of an art, with empirical considerations predominating. No comprehensive theory exists to guide the choice of scaling procedures. In the research to be described fitness scaling is employed when solving the Traveling Salesman Problem and not employed when solving 3-Satisfiability. the choice of whether to use fitness scaling or not is determined by the empirical evidence: the researcher tries some scaling algorithms, and if they improve performance enough to justify the extra cost incurred in their evaluation scaling is employed. Scaling is not necessary when solving 3SAT problems because the small and uniform nature of the fitness determinants (clauses with 3 variables apiece) is a very good measure of gene fitness. The terms fitness function value, fitness value and fitness will be used interchangeably in order to refer to the fitness score of a gene.

After each individual has been assigned a fitness value we need to select which individuals will contribute material to the next generation. We want the most fit individuals to have more offspring (*children*), but we also want to avoid premature convergence. There are two basic ways to do this. The first method is deterministic. In this method the individuals are ranked according to fitness, with individuals that are more fit receiving higher rank than less fit individuals. The number of offspring an individual produces is determined by rank, with higher-ranking individuals allowed to produce more offspring than lower-ranking ones. The second method is stochastic. In this method a roulette wheel is created with a number of slots equal to the sum of the fitness of each individual in the population. Each individual is assigned space proportional to its fitness on this imaginary roulette wheel. The roulette wheel is spun a number of times and the winners mate. This method gives all individuals a chance to mate, but still favors more fit individuals.

There are many different genetic operators available but the most widely used are crossover and mutation. Mutation is a unary operator, requiring as input one gene and returning one gene. In mutation each allele contained in a gene is changed with some low probability (typically around 0.0001) to the value of a different allele randomly chosen from A. Mutation is applied to all genes chosen to mate. The low probability of mutating any given allele means that many strings will undergo the process of mutation unchanged. Mutation ensures that no schema is ever lost permanently from the population, as a chance mutation might reintroduce it.

Crossover is a binary operator, requiring two genes as input and returning two genes. Two genes are chosen and a crossover point along the length of the genes is randomly picked. The first gene is copied until the crossover point, and the second gene is copied from then until the end of the gene. For example

$$001+110 \qquad \gamma_1$$

$$101+000 \qquad \gamma_2$$

after crossover at the position marked by $+$

$$001000 \qquad \gamma_1'$$

$$101110 \qquad \gamma_2'$$

The children γ_1' and γ_2' will join the next generation while the parents γ_1 and γ_2 will be discarded.

This type of crossover is known as one-point crossover because only one crossover point is chosen. In one point crossover alleles that are widely separated on the gene often end up on different children. This is known as *disruption*. Disruption is not desirable if alleles on opposite ends of the gene need to be correlated in order to solve the problem. For example, if the algorithm were trying to develop a plan which would enable a vehicle to maneuver in and out of a tight space in a minimal number of moves, it would be important for the last few moves to work together with the first few moves in order to allow the vehicle to back out of whatever spot it had pulled into. If the oppo-

site ends of the gene changed independently (because they were so often disrupted) the plan would be repeatedly frustrated, with two good halves that didn't fit together into a coherent whole. There have been several attempts to generalize crossover in order to solve the problem of disruption. Two point crossover treats the gene as a circle with the first and last alleles contiguous and chooses two crossover points. This lessens disruption at the ends of the gene but still tends to disrupt widely separated alleles. Two point crossover has been generalized to n-point crossover, where each allele has an equal chance to end up in either of the resulting children.

After the new children have been generated a new population P′ is formed. This can be done in two different ways. The first way is to generate n children and replace the entire old population P with the children from P′. The research to be described is a variation on this method. The second way, known as *elitism*, is to replace only a portion of the previous population P with children from P′. Typically, the best n/c parents are kept, with c a constant greater than 1, the remaining n-n/c population slots being filled with the n-n/c children produced during mating. In this way the best genes can remain in the population from generation to generation until replaced by even better children.

2.1 The Schema Growth Equation

Schema are bit patterns within genes that serve as sub-solutions to a problem. They are defined over $A \cup \{*\}$, in my research, A={0, 1}. The * is a wild card symbol used to match any symbol in A. For example, the schema *1* matches all genes of length 3 with a 1 in the second position: 010, 011, 110, and 111. We say 010, 011, 110 and 111 *contain* *1*. Given an alphabet A and gene length L the number of schema is $(|A|+1)^L$. For a binary alphabet, as used in the research to be described, the number of schema is 3^L. The usefulness of schema is that they allow us to view the success of a gene γ as a product of the schema that γ contains. A successful schema is one that represents a good sub-solution to the problem. In our previous example of trying to find a gene to

maximize the function x^2 the schema 1** is more fit on average than the schema 00* because the genes that 1** matches (100, 101, 110, 111) are more fit on average than the genes 00* matches (000, 001). We will say that a schema σ_1 is more fit than a schema σ_2 if, on average, the genes that contain σ_1 are more fit than the genes that contain σ_2.

Schema are the building blocks of genes. The *Building Block Hypothesis* states that good solutions result from good partial solutions, that is, good genes contain good schema. The Building Block Hypothesis is the major intuitive justification for the use of genetic algorithms in problem solving.

Since schema are the key to finding a good solution we would want confirmation that the genetic algorithm does indeed lead to the formation and propagation of good schema. There is an analysis, due to Holland, based on the k-armed bandit problem that shows the genetic algorithm is near-optimal in terms of schema processing. This analysis leads to the Schema Growth Equation. The key to this proof is that we want to test each schema enough times to know whether it is likely to be productive, but not so often as to waste our time on it.

Suppose we have a two-armed slot machine. We know one arm pays an award μ_1, with variance σ_1 and the other arm pays an award μ_2 with variance σ_2 and that $\mu_1 \geq \mu_2$ but we don't know which arm pays off more frequently. One way to maximize our gain is to perform some experimental trials to find the better arm and then stick with it until the end. If we have N trials to allocate to the two arms, we can choose to allocate n ($n<N/2$) pulls to each arm. We then allocate the remaining N-2n trials to the observed best arm. Our expected loss will then be

$$L(N,n)=|\mu_1-\mu_2|[(N-n)q(n)+n(1-q(n))]$$

where q(n) is the probability that the worst arm is the observed best arm after n trials. q(n) can be approximated by the tail of the normal distribution

$$q(n) \cong \frac{1}{\sqrt{2\pi}} \cdot \frac{e^{-x^2/2}}{x}$$

where

$$x = \frac{\mu_1 - \mu_2}{\sqrt{\sigma_1^2 + \sigma_2^2}} \cdot \sqrt{n}$$

Holland ([5]) shows a better way to allocate trials between the two arms. If we let n* be the number of trials we allocate to the observed worse arm, he obtains the following result

$$N - n^* \cong N \cong \sqrt{8\pi b^4 \ln N^2} \cdot e^{n^*/2b^2}$$

where

$$b = \frac{\sigma_1}{\mu_1 - \mu_2}$$

In short, we should try and allocate exponentially increasing numbers of trials to the observed better arm. Does the genetic algorithm do this?

The answer lies with the analysis of the schema present in a population [11]. We shall denote the situation where we have m examples of a schema σ in the population P at time t by $m(\sigma, t)$. Let $\bar{f}(\sigma)$ be the average fitness of all genes containing schema σ. Let $v(\gamma)$ be the fitness value of gene γ. Each gene γ_i gets selected with probability $\phi(\gamma_i)=v(\gamma_i)/\Sigma v(\gamma_j)$. After picking a non-overlapping population of size n with replacement from P we would expect

$$m(\sigma, t+1) = m(\sigma, t) \frac{n \bar{f}(\sigma)}{\Sigma v(\gamma_j)}$$

If we let $\bar{v}(P) = \frac{\Sigma v(\gamma_j)}{n}$ we obtain

$$m(\sigma, t+1) = m(\sigma, t) \frac{\bar{f}(\sigma)}{\bar{v}(P)}$$

Assume the fitness of a particular schema a in G remains above the average fitness of all schema s in G by an amount $c\bar{v}(P)$ with c a constant. We now have

$$m(\alpha, t+1) = m(\alpha, t)\frac{(\bar{v}(P)+c\bar{v}(P))}{\bar{v}(P)}$$

which is equal to

$$m(\alpha,t+1)=(1+c)m(\alpha,t)$$

Beginning with t = 0, we solve the recurrence relation to obtain

The Schema Growth Equation

$$\mathbf{m(\alpha,t)=m(\alpha,0)(1+c)^t}$$

This concludes our demonstration that the genetic algorithm does indeed allocate exponentially many trials to successful schema. Counterbalancing the genetic algorithm's inherent tendency to allocate exponentially increasing trials to successful schema is the use of crossover and mutation, which disrupt many schema in their exploration of the search space. Goldberg ([6]) derives a result confirming that in a population of size n on the order of n^3 schema are processed per generation. Each time the algorithm evaluates n genes it is usefully processing n^3 schema and allocating them a near optimal number of descendants. This result is known as *implicit parallelism.*

2.2 An Example of Schema Fitness

In this section we will look at a sample set of genes $\Gamma=\{\gamma_1, \gamma_2, ..., \gamma_n\}$ of length 3, the fitness $v(\gamma_i)$ of each γ_i the schema contained by the genes in Γ, the average fitness of a gene in G, $\bar{v}(\Gamma)$ (which is equal to $\bar{f}(*,*,*)$), and $\bar{f}(\sigma)$ for all schema σ. In our example all genes will be of length 3 and each allele will come from {0, 1}. Schema are necessarily the same length as the genes they describe, and each schema is a member of $\{0, 1, *\}^3$.

See tables 2.1 and 2.2 on the following pages for sample calculations.

Table 2.1 Each gene γ and its corresponding fitness value $v(\gamma)$

γ	$v(\gamma)$
000	5
001	4
010	4
011	3
100	6
101	5
110	5
111	4

Table 2.2 Calculation of schema fitness from the fitness of each gene

σ	$\bar{f}(\sigma)$	γ_1	γ_2	γ_3	γ_4	γ_5	γ_6	γ_7	γ_8
***	4.5	5	4	4	3	6	5	5	4
**0	5	5		4		6		5	
**1	4		4		3		5		4
0	5	5	4			6	5		
*00	5.5	5				6			
*01	4.5		4				5		
1	4			4	3			5	4
*10	4.5			4				5	
*11	3.5				3				4
0**	4	5	4	4	3				
0*0	4.5	5		4					
0*1	3.5		4		3				
00*	4.5	5	4						
000	5	5							
001	4		4						
01*	3.5			4	3				
010	4			4					
011	3				3				
1**	5					6	5	5	4
1*0	5.5					6		5	
1*1	4.5				5				4
10*	5.5					6	5		
100	6					6			
101	5						5		
11*	4.5							5	4
110	5							5	
111	4								4

2.3 Representation Issues

Most of the art in designing a genetic algorithm consists of finding a good representation. A good representation should be:

- Isomorphic to the problem. Anyone who is familiar with the problem structure should be able to straightforwardly identify which part of the representation stands for what part of the problem.
- Efficient to implement and run. An inefficient representation only obscures the progress made by the algorithm behind hours of waiting around for the overhead processing to end.
- Useful with respect to the genetic operators chosen. For example, if one-point crossover and mutation are the operators chosen, the representation should use structures for which one-point crossover and mutation tend to produce superior solutions.

The following sections detail problems that can arise in genetic representation design, introduce terminology to classify different types of difficulties and propose a method to overcome them.

2.3.1 Drowning

When the number of possible genes is much greater than the number of valid solutions to the problem at hand we call the representation a *drowning* representation. Given an alphabet A from which alleles are chosen and gene length L there are $|A|^L$ different possible genes. Each one of these genes can be produced by some combination of genetic operations. This is a desirable feature in problems where every possible gene is also an allowable solution. On the other hand, if the problem is highly constrained, most of the possible genes will be invalid solutions. We present a Traveling Salesman Problem representation as an example. Let each allele stand for any city. If there are 10 cities our tour will be of length 10, and our genes will be 10 alleles long.

There are therefore 10^{10} different possible genes. On a fully connected graph there are at most 10! different paths, so we have many more possible genes than allowable paths.

There are three standard approaches to a representation that is drowning. The first approach is simply to ignore the invalid genes. This can be accomplished by giving invalid genes very low fitness values. This may work if the representation doesn't produce too many invalid genes.

The second approach is to save the representation by throwing it a life preserver in the form of a many-to-one mapping between genes and valid solutions. This mapping will be an algorithm that converts invalid genes into valid solutions. The problem with such an approach is that the isomorphism between the representation and the problem is lost. The gene no longer codes for a solution, but is input to an algorithm that produces a solution from it, perhaps introducing unintended errors. Genes that code for the same solution may have different values at many alleles and crossover will produce children that don't resemble either parent. There is also no guarantee that the same position in each gene will code for the same subpart of the problem. This invalidates the Building Block Hypothesis for this representation because it is impossible to determine which schema are successful if schema mean different things in different genes

The third approach is to change the representation. This removes all the difficulties associated with the drowning representation but adds new problems, most importantly the need to throw out most of the work already done and start from scratch. We would like to avoid this "solution" if at all possible.

In section 2.3.3 below will introduce a solution to the above problems called the *target problem* solution.

2.3.2 Blindness

A representation is *blind* if genes that are valid solutions tend to have a high proportion of children which are invalid solutions, or whose children are worse solutions. The previous example of the maneuvering vehicle representation is also a blind repre-

sentation because any gene γ that coded for a good series of maneuvers (move in, maneuver around, back out, all without hitting anything) would most likely mate with a gene γ' coding for a series of moves that would lead to collision with an obstacle when combined with the series of moves in γ. This would result in a child that coded for an invalid solution. Coding the problem in that fashion would lead to a situation in which genes are tried out more or less at random with the program groping in the dark for good genes.

Blindness results from a mismatch between the representation and the genetic operators. The operators used should be tailored so as to pick out the important parts of the representation to be passed on to future generations. Problems with multiple optimum (or near optimum) solutions tend to lead to blindness. The algorithm has a tendency to mix the alleles from genes that are good approximations to one optima with alleles from genes that are good approximations to another optima, resulting in genes that aren't good approximations to any optima.

The next section will introduce a solution to these problems, called the *target problem* solution.

2.3.3 The target problem solution

The solution to both drowning and blindness is to adopt a *target* problem for which a good representation is known. The original problem is then translated into an instance of the target problem and the algorithm attempts to solve that instance. When the instance of the target problem is solved all that remains is for that solution to be translated back into a solution for the original problem.

A major goal of genetic algorithm research should not be to find new and trickier representations for problems that don't lend themselves to straightforward encoding, but to build a library of well-understood target problems. A researcher faced with a new problem to be solved could look up her problem in a standard reference book and determine if it were a standard target problem, or if a transformation were known to a

standard target problem. Even if no transformation were known it would often be easier to find one than to devise a complicated encoding scheme with no guarantee of success. Researchers today are in much the same position complexity theorists would be without NP-completeness reference guides. Every new problem means starting from scratch.

A major goal of my research aims at making 3SAT a standard target problem. An elegant encoding scheme is described and an efficient set of operators is used. Drowning is avoided because all possible bit strings are also valid trial solutions. Blindness is avoided because the structure of the problem (a great number of small clauses, each of which influences the fitness of the gene slightly) lends itself to effective search given the geneitc operators one-point crossover and mutation. As shown in the empirical results good parents give rise to good offspring. In addition, overpopulation is introduced as a method of speeding up the propagation of good schema in successive generations and is used with the new fitness estimation procedure bitwise expected value in order to drastically reduce the time to solution. Overpopulation and bitwise expected value are demonstrated to work well on 3SAT and both have wide applicability when 3SAT is considered as a target problem for a wide class of real world applications. The use of overpopulation and bitwise expected value is by no means limited to just Satisfiability problems, however.

3. Bitwise Expected Value

This section introduces bitwise expected value (bev), a method which estimates the fitness of a gene γ given the fitness values of γ's parents and the number of alleles each parent has contributed to γ during mating. Bitwise expected value is used in conjunction with the method of overpopulation, introduced in the next section, to reduce time to solution. Empirical evidence will be presented showing that bitwise expected value is highly correlated to the actual fitness function value on 3SAT problems. In chapter 4 empirical evidence will be presented comparing a genetic algorithm which uses bev and overpopulation to two other GAs: an algorithm which uses neither bev nor overpopulation, and an algorithm which uses overpopulation and a different fitness estimation function. Bitwise expected value is an excellent method for the prediction of a gene's fitness value as it is computationally inexpensive and uses a minimal amount of information.

3.1 Definition

Given a child gene γ with parents p_1 and p_2 where p_i has fitness value $v(p_i)$, length l_i, and contributes n_i alleles to γ, the bitwise expected value of γ is given by

$$bev(\gamma) = n_1 \frac{v(p_1)}{l_1} + n_2 \frac{v(p_2)}{l_2}$$

We call $\pi(\gamma)$ = (v(γ)/length of γ) the *per-bit* fitness of γ.

3.2 Example Calculation of Bitwise Expected Value

In Table 3.1, below, a sample population of genes of length 6 with alleles drawn from A={0, 1} is shown. Each row contains the name of the gene γ, the gene itself, and its fitness value v(γ). Table 3.2 contains the children produced by crossover on a sample run, their parents, the crossover point, the amount of the child's bev that each parent

has contributed, and the child's bev. As crossover is a binary operator each child has a pair of parents and each pair of parents creates two children. For example, when γ_0 undergoes crossover with γ_1 at position 3, two children are produced. The first child ($child_0$) receives the allele values for its first three positions from γ_0 and the allele values for its three remaining positions from γ_1. Its bitwise expected value is therefore $3(v(\gamma_0)/6)+3(v(\gamma_1)/6)=6.0+7.5=13.5$. The second child produced ($child_1$) receives the allele values for its first three positions from γ_1 and the allele values for its three remaining positions from γ_0. Its bev is therefore $3(v(\gamma_1)/6)+3(v(\gamma_0)/6)=7.5+6.0=13.5$. This calculation is equivalent to π(first parent)*(the number of alleles the first parent has contributed)+π(second parent)*(the number of alleles the second parent has contributed). That both children have the same bev in this example is coincidental. In general the children's bevs will be different, although they are constrained to lie between the parent's fitness values.

Please see tables 3.1 and 3.2 on the following page for example bitwise expected value calculations on a sample population. Parents were selected by hand for purposes of this example.

3.3 Conditions for Effective Use of Bitwise Expected Value

For bev to be useful it is necessary that it be a very good approximator of the actual fitness function. Bitwise expected value is intuitively appealing on this count because it assumes that the fitness of a gene γ is a product of the fitness of the individual alleles γ contains. Bev can be seen as a parallel of the Building Block Hypothesis. In the Building Block Hypothesis the fitness of a schema σ is the sum of the fitness values of all the genes that contain σ divided by the number of genes that contain σ. In bev the fitness of an allele a is the fitness of the gene γ which contains a divided by the number of alleles in γ. The fitness value of a gene γ is a function of the fitness values of the schema that γ contains. The bev of a gene γ is the sum of the per-bit fitnesses of the alleles that γ contains.

3.4 Empirical Results

See table 3.3 for data demonstrating that bitwise expected value is very highly correlated with the fitness function. L is the length of the chromosome, R is Pearson's R statistic, N is the number of bev-fitness function pairs examined and P is the probability of such a correlation happening by chance. $P<0.05$ is usually considered significant. Also included as graph 3.1 is a scattergraph of a sample run as a graphic illustration of the correlation. L for this example is 30, R is 0.9692 and N is 2880. The pairs examined are obtained as follows. Every gene in the temporary population has its bev calculated. Those genes which are chosen to enter the population for the next generation also have their fitness calculated. The bev-fitness pairs are the bev and fitness of these genes. These values are taken from actual runs, as opposed to the data in tables 3.1 and 3.2, which are for example purposes only.

3.5 Effect on Parallelization

Bitwise expected value is a local method. All information required for the calculation of bev is present at the time crossover is performed: the parents' fitness values, their lengths, and how many alleles each parent contributes to the child. Therefore, when crossover is performed in parallel over a population, the calculation of bev requires no extra serial processing. Bitwise expected value was implemented in MPL on a MasPar DPU Model MP-1208 running Ultrix V4.2A. No extra code was needed in order to port bev from a serial implementation in C under UNIX 4.3BSD to a parallel implementation in MPL under Ultrix. The average number of generations to solution was identical in the serial and parallel implementations, as expected. The advantage of a parallel implementation lies not in reducing the average number of generations to solution but rather in the simultaneous application of the fitness function to every gene in the population. The serial application of the fitness function to each gene in turn creates a bottleneck in serial implementations. When the population is large parallel application of the fitness function can greatly reduce execution time over serial application as

Table 3.1 Each gene γ in the sample population and its associated fitness value $v(\gamma)$

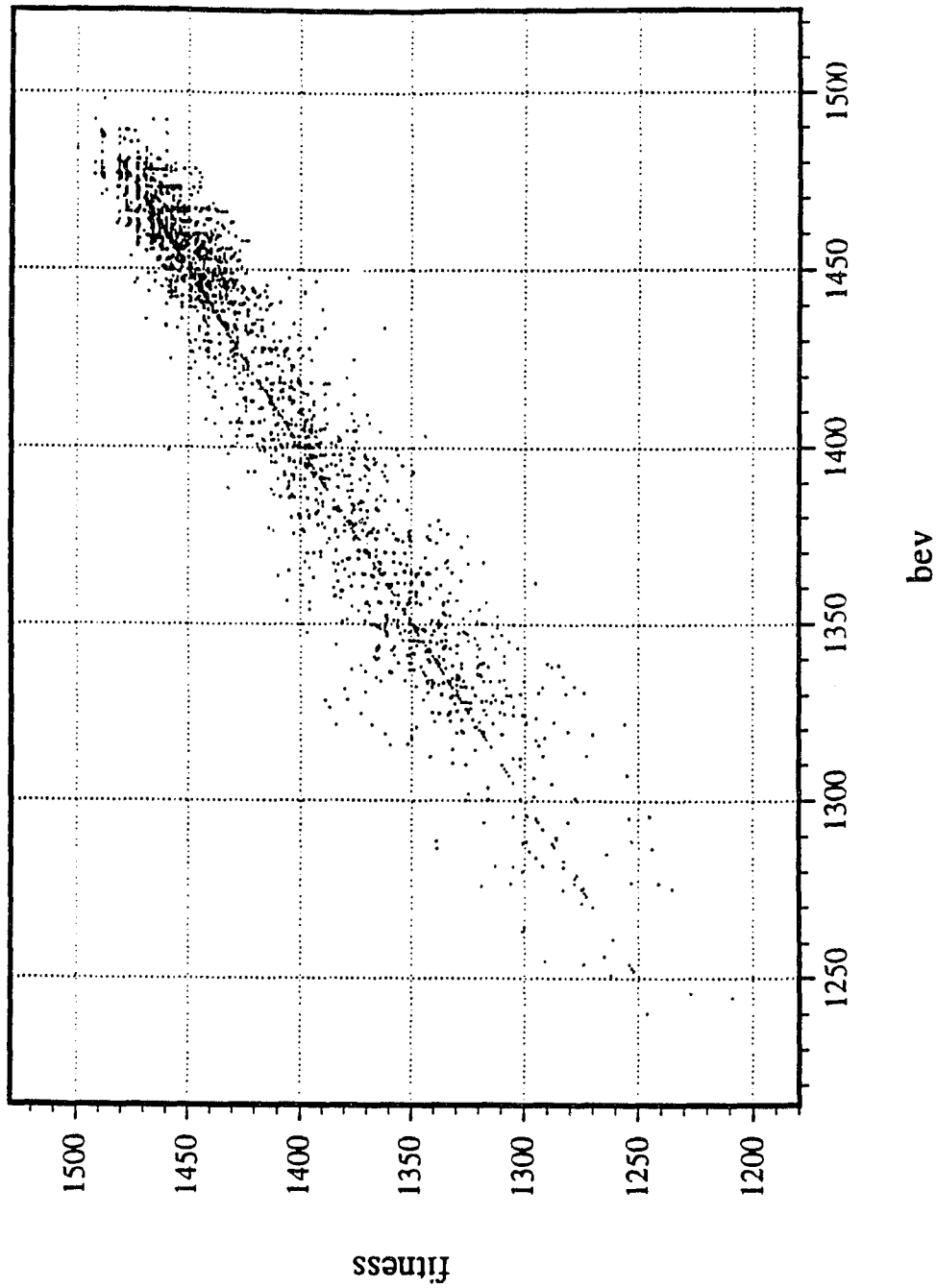
	γ	$v(\gamma)$
g_0	111010	12
g_1	010111	15
g_2	100100	6
g_3	011010	18
g_4	101100	21
g_5	000001	5
g_6	100001	13
g_7	010101	12

Table 3.2 Calculation of bev for each of the resulting children

		parents	crossover point	$fitness_0$	$fitness_1$	bev
$child_0$	111111	0,1	3	6.0	7.5	13.5
$child_1$	010010	0,1	3	7.5	6.0	13.5
$child_2$	011010	7,3	1	2.0	15.0	17.0
$child_3$	010101	7,3	1	10.0	3.0	13.0
$child_4$	010100	1,2	4	10.0	2.0	12.0
$child_5$	100111	1,2	4	5.0	4.0	9.0
$child_6$	011010	1,3	2	5.0	12.0	17.0
$child_7$	010111	1,3	2	10.0	6.0	16.0

Table 3.3 Correlation between bitwise expected value and the fitness function

length = 10		length = 20		length = 30		length = 50	
r	N	r	N	r	N	r	N
0.9294	560	0.9575	1440	0.9562	1800	0.9765	6200
0.8894	200	0.9575	1440	0.9562	1800	0.9765	6200
0.8777	240	0.9550	1120	0.9678	2640	0.9836	6800
0.9061	240	0.9316	880	0.9698	2760	0.9800	6400
0.9460	440	0.9488	1440	0.9700	2400	0.9781	7600
0.8688	80	0.9396	1040	0.9705	3240	0.9795	7000
0.9373	960	0.9492	1200	0.9677	2880	0.9797	6600
0.9258	600	0.9601	1840	0.9674	2880	0.9830	7200
0.8920	360	0.9050	800	0.9637	2520	0.9761	5800
0.9350	160	0.9585	1200	0.9692	2880	0.9805	7000
0.9367	520	0.9541	1120	0.9689	2040	0.9781	5800
0.9010	280	0.9403	1120	0.9568	2400	0.9736	3800
0.8043	160	0.9170	880	0.9623	2040	0.9789	6200
0.9000	360	0.9633	1520	0.9662	2880	0.9832	7600
0.9208	400	0.9464	1360	0.9741	3480	0.9787	7600
0.9115	200	0.9423	1200	0.9659	2400	0.9801	7000
0.9117	320	0.9312	1040	0.9750	2400	0.9788	7600
0.9358	240	0.9218	640	0.9670	2640	0.9778	6800
0.9211	520	0.9400	1200	0.9680	3000	0.9793	5600
0.8941	240	0.9665	1120	0.9617	2520	0.9810	6600



Graph 3.1 Bev vs. fitness

the most expensive phase of a genetic algorithm is usually the evaluation of the fitness function. The calculation of bev puts no restriction on the parallelization of a genetic algorithm. Bev can be computed for each gene in parallel in the same manner as the fitness function

3.6 Future Directions

There are two basic directions applications of bitwise expected value can explore in the future. The first direction deepens the relationship between bev and genetic algorithms. The second direction applies bev to other learning and optimization methods.

3.6.1 Future directions for bev within the GA paradigm

The form of bitwise expected value described above assumes that each allele in γ is equally important in estimating the fitness function value of γ so a weighted average is used. This can be generalized by assigning varying weights to individual positions along γ. Given genes of length L, $W = (w_1, w_2, ..., w_L)$ is an n-tuple of weights where weight w_i is assigned to position i. There are two different ways to assign weights to positions: static and dynamic. Static methods incorporate problem knowledge into the calculation of bev through the initial choice of W. Dynamic methods incorporate knowledge of how the environment is changing over time into changes in W.

The static method differentially weights each position's contribution to the child's bev according to a formula decided upon before execution of the algorithm. Any formula can be used to determine W. If some domain information is available that indicates that the alleles which occur at one or more positions account for a higher proportion of a gene's fitness than others W can be modified by weighting those more important positions more than other, less important, positions. Given child gene γ, parents p_1 and p_2, gene length L, weight L-tuple W, crossover point $0<k<L$, and the definitions from section 3.1, we obtain the more general formula

$$bev(\gamma)=(\pi(p_i)w_1)+(\pi(p_1)w_2)+...+(\pi(p_1)w_k)+(\pi(p_2)w_{k+1})+...+(\pi(p_2)w_L)$$

Bev can also easily be extended to handle n-point crossover by replacing the sum shown above by summation over an arbitrary number of crossover points. Similarly for those genetic operators which involve an arbitrary number of parents, the contribution each parent p makes to γ can be calculated by multiplying the per-bit value of p by the number of alleles p contributes to γ.

As currently implemented bev contains no penalty for mutation but one can easily be added by setting the child's bev equal to the fitness value of the single parent p, and then subtracting $\pi(p)$ from the child's bev for each mutation that occurs to p. This would embody the assumption that mutation is likely to be harmful. It would also be possible to make the subtraction of each $\pi(p)$ probabilistic. A child γ whose parents have low fitness values would have a lower probability of having $\pi(p)$ subtracted from its bev for each mutation of p than a child γ' whose parents have high fitness values. This would allow for the possibility that applying mutation to less fit genes is less likely to be harmful than applying mutation to highly fit genes.

A dynamic method would perform an ongoing analysis of the performance of bev as a predictor of the fitness function. The bev of a gene γ would be compared to the fitness value of γ to measure the accuracy of bev as a predictor of the fitness function with the current W. If bev loses its ability to accurately predict fitness a dynamic method would change W. There are many methods which can serve as guidelines for the modification of W, including statistical methods such as analysis of variance or regression analysis, but the most natural method would be to vary the weights in W at random and search using genetic techniques for a W' which yields good results.

3.6.2 Future applications of bev in other paradigms

Many different variants on bitwise expected value can be developed for different learning and optimization techniques. All that is needed for the calculation of bev is a group of parents, the fitness values of those parents, their sizes, and how much material has been taken from each to form the child(ren).

For example, classifier systems ([45]) are a variant of genetic algorithms in which if-then rules are encoded as genes. We name the variant of bitwise expected value that is applicable to classifier systems *bitwise expected productivity* (bep). We calculate bep in the same manner as bev except that production rules take the place of genes.

4. Overpopulation

A standard genetic algorithm has a preset population size limit of n genes. During reproduction enough genes are chosen to mate from generation t so that the next generation t+1 will also consist of n genes. In overpopulation a multiplication factor M>1 is chosen and Mn genes are produced during mating, resulting in more genes than the population can hold. These Mn genes are assigned to a temporary population T. A fitness estimation function is applied to all genes in T and the n genes with the highest estimated fitness values become the next generation. The fitness estimation function used to evaluate genes in T need not be the same fitness function $v(\gamma)$ that evaluates genes as candidate solutions. The fitness estimation function should be efficient to calculate because it is evaluated over Mn strings every generation, many more evaluations than the actual fitness function. It should also be strongly positively correlated with $v(\gamma)$ in order to be a good predictor of the child's actual fitness value. In my research bitwise expected value was used as the fitness estimation function. Empirical results show very strong correlation between the bitwise expected value of a gene and its actual fitness value, as shown in Chapter 3.

4.1 Definition

The population size in a standard genetic algorithm is set before the execution of the algorithm to an amount that the designer thinks will lead to effective search. Population size is usually changed a few times, in between executions, in the process of tuning the algorithm. In the tuning process tradeoffs are made between exhaustiveness of search (large n) and speed of convergence (small n). When genes are chosen from generation t for reproduction enough genes are chosen so that exactly n children will result, thus forming generation t+1. For example, when the population size is 10 and crossover is the only genetic operator in use, 10 genes are chosen, with replacement, to undergo mating. Crossover requires two genes as input and returns two new genes as output so

the 10 chosen genes will be paired up for crossover. Each pair of genes will yield two new genes which will be members of the next generation. This maintains a constant population size of 10 genes. In overpopulation a multiplier M is chosen, M>1, and enough genes are chosen to mate from generation t to form a temporary population T of size Mn. For example, if M=2 then 20 genes are chosen for crossover and 20 children result.

Even though there are now Mn>n children in T the population size of generation t+1 does not increase. Instead, each child has its fitness function value estimated using a fitness estimation function and the children are sorted according to their estimated fitness value. The n children with the highest estimated fitness values become generation t+1 and the rest are discarded.

We will use the population from table 3.1 to form the descendants in the temporary population T as shown in table 4.1. The next generation is given in table 4.2.

4.2 The Use of Different Estimation Functions

Bitwise expected value is an attempt to ensure that large schema present in the more successful parents survive in their children, as large chunks inherited from good parents lead to high bitwise expected values in their children. Other estimation functions may concentrate on different aspects of the parents to estimate the children's fitness values. Overpopulation will improve the performance of any genetic algorithm when used with any estimation function that is positively correlated with the fitness function, although it of course works best using those functions which are more highly correlated with the fitness function.

Table 4.1 Calculation of bev for each of the children in the temporary population

		parents	crossover point	$fitness_0$	$fitness_1$	bev
$child_0$	111111	0,1	3	6.0	7.5	13.5
$child_1$	010010	0,1	3	7.5	6.0	13.5
$child_2$	011010	7,3	1	2.0	15.0	17.0
$child_3$	010101	7,3	1	3.0	10.0	13.0
$child_4$	010100	1,2	4	10.0	2.0	12.0
$child_5$	100111	1,2	4	4.0	5.0	9.0
$child_6$	011010	1,3	2	5.0	12.0	17.0
$child_7$	010111	1,3	2	6.0	10.0	16.0
$child_8$	111011	0,7	5	10.0	2.0	12.0
$child_9$	010100	0,7	5	10.0	2.0	12.0
$child_{10}$	000100	1,2	1	2.5	5.0	7.5
$child_{11}$	110111	1,2	1	1.0	12.5	13.5
$child_{12}$	101010	4,3	3	10.5	9.0	19.5
$child_{13}$	011100	4,3	3	9.0	10.5	19.5
child14	101101	4,7	4	14.0	4.0	18.0
$child_{15}$	010100	4,7	4	8.0	7.0	15.0

Table 4.2 The resulting population after being sorted according to bev

		parents	$fitness_0$	$fitness_1$	bev
$child_1$	010010	0,1	7.5	6.0	13.5
$child_2$	011010	7,3	2.0	15.0	17.0
$child_6$	011010	1,3	5.0	12.0	17.0
$child_7$	010111	1,3	6.0	10.0	16.0
$child_{12}$	101010	4,3	10.5	9.0	19.5
$child_{13}$	011100	4,3	9.0	10.5	19.5
child14	101101	4,7	14.0	4.0	18.0
$child_{15}$	010100	4,7	8.0	7.0	15.0

4.3 Generalization of the Schema Growth Equation to Include Overpopulation

In this section the schema growth algorithm presented in section 1.2 is generalized, demonstrating that the use of overpopulation leads to the creation of more genes containing above average schema than would result without the use of overpopulation.

4.3.1 Preliminary notation

We will use the notation given in section 1.2, above, with the following additions:

Let $v(\gamma)$ be the fitness of a gene γ.

Let $\bar{v}(\{\gamma_1, \gamma_2, ..., \gamma_n\})$ be $\Sigma v(\gamma_i) / n$.

Let $d(\Gamma)$ be the distribution of the fitness values of the genes in the population as a whole.

Let $d(\sigma)$ be the distribution of the fitness values of all genes containing schema σ.

Let $\bar{f}(\sigma)$ be the average fitness of all genes containing schema σ.

Let $|\sigma|$ be the number of genes containing σ.

Let $d(\bar{\sigma})$ be the distribution of the fitness values of all genes not containing schema σ.

Let $\bar{f}(\bar{\sigma})$ be the average fitness of all genes not containing schema σ.

Let $| \bar{\sigma} |$ be the number of genes not containing σ.

Let $p(\sigma, x)$ be the probability of picking a gene γ containing schema σ such that $v(\gamma)>x$.

Let $p(\bar{\sigma}, x)$ be the probability of picking a gene γ which does not contain schema σ such that $v(\gamma)>x$.

Let $\lambda = \{\gamma \mid v(\gamma) = \min\{v(\gamma_1), v(\gamma_2), ..., v(\gamma_n)\}$. We will call λ by the name λ' when λ does not contain σ.

Let $\omega = v(\lambda)$ and $\omega' = v(\lambda')$.

4.3.2 Statement of the equation

We will derive $m'(\alpha, t)$ such that:

$$m'(\alpha,t) = m'(\alpha,0)\,(1+c)^t + (t \cdot \rho(\alpha)) \;>\; m(\alpha,t) = m(\alpha,0)\,(1+c)^t$$

where $m(\alpha, t)$ is the schema growth equation from section 1.2, α is an above average schema, c is a constant, and $\rho(\alpha)$ is the probability that overpopulation will add an extra gene containing α to the population at time t. For the above inequality to hold we must show that $\rho(\alpha)>0$.

4.3.3 The derivation of the generalized schema equation

We know from section 1.2 above that when the genetic algorithm generates n new genes $\Gamma = \{\gamma_1, \gamma_2, ..., \gamma_n\}$ to form generation t+1 the expected number of strings in generation t+1 containing σ is

$$m(\sigma,t+1) = m(\sigma,t)\,\frac{n\bar{f}(\sigma)}{\Sigma v(\gamma_j)}$$

Consider the case where we use overpopulation with multiplication factor M=(n+1)/n and therefore we generate n+1 new genes $\{\gamma_1, \gamma_2, \gamma_n, \gamma_{n+1}\}$. Our new equation is

$$m'(\sigma,t+1) = m'(\sigma,t)\,\frac{n\bar{f}(\sigma)}{\Sigma v(\gamma_j)} + \rho(\sigma)$$

The best n genes are kept to make up the new generation t+1. It is easy to see that gene γ_{n+1} will only be one of the top n genes if $v(\gamma_{n+1})>\omega$. This leads to Equation 4.1, the definition of $\rho(\sigma)$ as

the probability that γ_{n+1} contains σ and λ does not contain σ and the fitness of γ_{n+1} is greater than the fitness of λ.

minus

the probability that γ_{n+1} does not contain σ and λ does contain σ and the fitness of γ_{n+1} is greater than the fitness of λ

$$= \quad m'(\sigma,t)\frac{\bar{f}(\sigma)}{\sum v(\gamma_j)} m'(\bar{\sigma},t)\frac{\bar{f}(\bar{\sigma})}{\sum v(\gamma_j)} p(\sigma,\omega') - m'(\bar{\sigma},t)\frac{\bar{f}(\bar{\sigma})}{\sum v(\gamma_j)} m'(\sigma,t)\frac{\bar{f}(\sigma)}{\sum v(\gamma_j)} p(\bar{\sigma},\omega)$$

$$= \quad m(\sigma,t)\frac{\bar{f}(\sigma)}{\sum v(\gamma_j)} m(\bar{\sigma},t)\frac{\bar{f}(\bar{\sigma})}{\sum v(\gamma_j)} (p(\sigma,\omega') - p(\bar{\sigma},\omega))$$

Equation 4.1

If no gene in the population contains σ, then it is impossible for overpopulation to introduce one. That is the task of the standard genetic operators. If all genes in the population contain σ, then overpopulation cannot improve on the standard algorithm as there is no way to introduce more genes containing σ. We can therefore assume

$$m'(\sigma,t) > 0 \text{ and } m'(\bar{\sigma},t) > 0$$

Equation 4.1 reduces to the following description

the probability of the fitness value of a gene containing σ being greater than the fitness value of a gene (λ') not containing σ

minus

the probability of the fitness value of a gene not containing σ being greater than the fitness value of a gene (λ) containing σ.

Which can be stated symbolically

$$p(\sigma,\omega') - p(\bar{\sigma},\omega) > 0$$

Let m $=\bar{\nu}$ (Γ). By the Central Limit Theorem [36] we know d(Γ), d(s) and d($\bar{\sigma}$) are normally distributed for large sample sizes. We will use this fact in the following deri-

vations. In particular, we can assume w = $\bar{f}(\sigma)$ and w' = $\bar{f}(\bar{\sigma})$ as they are the fitness values of randomly selected genes. Let a be an above average schema in generation t.

"α is an above average schema" means

$$\bar{f}(\alpha) > \bar{f}(\Gamma)$$

Equivalently, by the Central Limit Theorem

$$p(\alpha,\bar{f}(\Gamma)) > 0.5$$

By definition

$$\bar{f}(\Gamma) = \frac{|\alpha|\bar{f}(\alpha)+|\bar{\alpha}|\bar{f}(\bar{\alpha})}{|\alpha|+|\bar{\alpha}|}$$

Therefore

$$(|\alpha| + |\bar{\alpha}|)\bar{f}(\Gamma) = |\alpha|\bar{f}(\alpha) + |\bar{\alpha}|\bar{f}(\bar{\alpha})$$

Equivalently

$$|\bar{\alpha}|\bar{f}(\Gamma) + (|\alpha|\bar{f}(\Gamma) - |\alpha|\bar{f}(\alpha)) = |\bar{\alpha}|\bar{f}(\bar{\alpha})$$

We know

$$\bar{f}(\alpha) > \bar{f}(\Gamma)$$

Therefore

$$|\alpha|\bar{f}(\alpha) > |\alpha|\bar{f}(\Gamma)$$

Let x = $|\alpha|\bar{f}(\alpha) - |\alpha|\bar{f}(\Gamma)$. We know x>0.

We obtain

$$|\bar{\alpha}|\bar{f}(\Gamma) - x = |\bar{\alpha}|\bar{f}(\bar{\alpha})$$

Therefore

$$\bar{f}(\Gamma) > \bar{f}(\bar{\alpha})$$

By the Central Limit Theorem

$$p(\bar{\alpha},\bar{f}(\Gamma)) < 0.5$$

As d(α) and d($\bar{\alpha}$) are normal distributions and the mean of d(α) is greater than the mean of d($\bar{\alpha}$) and ω>ω′ we can conclude

$$p(\alpha,\omega') - p(\bar{\alpha},\omega) > 0$$

This concludes our proof that ρ(α,t)>0.

To continue with our derivation of m′(α, t), we can skip the intermediate steps and obtain

$$m'(\alpha,t+1)=m'(\alpha,t)(1+c)+\rho(\alpha)$$

Beginning with t = 0 we solve the recurrence relation and obtain

The generalized schema growth equation

$$m'(\alpha,t)=m'(\alpha,t)(1+c)^t + \frac{\rho(\alpha)((1+c)^t - 1)}{c}$$

4.3.4 Discussion of the result

If an algorithm is to be called a genetic algorithm, and not merely a biologically inspired search heuristic it must allocate exponentially more trials to the observed better schema and it must generate new schema to test through the mating process.

Although many of the genetic algorithms currently used for function optimization and machine learning contain features that render the schema growth equation only approximately correct, the idea of allocating more trials to successful schema remains a constant. It will be noted that the above derivation did not depend upon the exact form of m(α, t). The only time m(α, t) and ρ(α) are combined is in the solution of the final recurrence relation. Despite surface variations all genetic algorithms allocate trials to schema based upon a recurrence relation, each time step representing one generation.

Only the form of $m(\alpha, t)$ differs from implementation to implementation. This is due to the variety of non-standard techniques which can be used to optimize the performance of the genetic algorithm in the solution of various problems.

A genetic search consists of two parts: a population and a set of genetic operators. Given a population P at time t, we will denote by P(t) the set of genes in population P at time t. We will say that P(t+1) is a *direct descendant* of P(t) if the available genetic operators can be applied to the genes in P(t) so as to result in P(t+1) in one generation. We say that P(t+n) is a descendant of P(t) if there exists a series of populations (P(t), P(t+1), P(t+2), ..., P(t+n)), such that P(t+j) is a direct descendant of P(t+(j-1)) for all $1 \leq j \leq n$. In order for a genetic search to be a genetic algorithm it is required that, given any population P(t), there must exist a descendant of P(t) which contains the desired solution.

Operators that introduce new schema vary from implementation to implementation but the above derivation only assumes that they exist, not that they take any particular form. If the solution of a problem requires a given schema σ, and σ is not present in the initial population, or is randomly eliminated at some time t, and the implementation cannot introduce new schema, then the implementation is not an algorithm because σ cannot be reintroduced in order to be part of the solution.

Genetic algorithms are still in their infancy and so it is hard to make any broader statements about their properties than I just have: that all GAs allocate exponentially more trials to successful schema, and that they do so by recombining successful genes from generation to generation.

Nevertheless, the result of using overpopulation can be stated swiftly and powerfully: If the Building Block Hypothesis is true for the problem under consideration, addition of overpopulation to a genetic algorithm G will lead to shorter search times irregardless of any additional features that have been added to G. Overpopulation can improve the performance of any genetic algorithm because its ability to fine tune the

growth rate of above average schema leads to the introduction of more successful schema than an algorithm without overpopulation would otherwise introduce.

4.4 Effect on Parallelization

Overpopulation does require some extra processing beyond the standard algorithm. This extra processing involves the overhead of keeping a larger temporary population T, application of the fitness estimation method to each gene in T, and the need to sort the children in T by estimated fitness value. The larger temporary population can be spread across the processors, as the application of the fitness estimation method can take place in parallel. There are many well-known parallel sorting techniques, so finding the n genes with the highest estimated fitness values is not an obstacle to either implementation or efficiency. The production of Mn instead of n genes requires more processors but the selection and reproduction of the genes can be done in parallel. The only inter-processor communication occurs at the beginning of reproduction, in order to determine which strings will undergo crossover, and at the end of the reproduction phase in order to return the children produced to the main population.

4.5 Future Directions

The most interesting variant of overpopulation would be to change the value of M during the execution of the algorithm. M can be used as a form of simulated annealing parameter. M can be set to a small value at the beginning of the run to allow a greater area of the search space to be explored. As the run progresses M can be gradually increased in order to favor the children of more successful individuals, thereby limiting the search to a smaller area. When an unsatisfactory equilibrium is reached the mutation rate can be raised or random strings introduced in order to explore distant areas of the search space.

Another interesting avenue is static optimization. Different problems may have qualities which call for different values of M. One of these qualities involves the *fitness*

landscape of the problem. Given a function f with two parameters x and y we can construct the fitness landscape of f(x, y) in much the same way we construct a terrain map of the Earth. The parameters x and y represent the coordinates of a point on the map and f(x, y) corresponds to the height of that point. The search for the minimum value of f(x, y) then corresponds to finding the lowest point in the fitness landscape. Fitness landscapes that are mostly flat require the algorithm to take advantage of even the smallest differential in fitness value between two genes. A flat landscape calls for a higher value of M, which will keep the algorithm focused on the most fit string and its descendants. A very hilly fitness landscape can trap an algorithm into thinking it has found the minimal solution when in fact it is merely in a deep crevasse. These crevasses are known as local minima and are frequently encountered when gradient descent methods are being used to explore the search space. Exploring a hilly fitness landscape requires sampling trial solutions from many different areas of the search space. A smaller value of M allows the algorithm to explore a larger section of the search space because the children of genes that initially appear less promising are less likely to be crowded out by the children of genes that initially appear more promising.

5. Using a GA to solve 3SAT

As a testbed for the new genetic methods bitwise expected value and overpopulation, we have chosen a problem which is challenging enough to test the new methods, is new to the literature, and has ramifications for the solution of an important class of related problems. The class of problems known as *NP-complete* is an important source of difficult problems for optimization and learning algorithms. We decided that 3-Satisfiability, or 3SAT, a problem well-known to be NP-complete, would be a good proving ground for our new methods.

5.1 3SAT

We follow [28] in our definition of 3SAT. Let $U=\{u_1, u_2, ..., u_m\}$ be a set of boolean *variables*. A *truth assignment* for U is a function $t:U\rightarrow\{T, F\}$. If t(u)=T we say that u is "true" under t; if t(u)=F we say u is "false" under t. If u is a variable in U, then u and $\bar{u}$ are *literals* over U. The literal u is true under t iff variable u is true under t; the literal $\bar{u}$ is true under t iff the variable u is false under t.

A *clause* c over U is a set of literals of size 3, such as $c= \{u_1, \bar{u}_3, u_8\}$. It represents the disjunction of those literals and is satisfied by a truth assignment iff at least one of its members is true under that assignment. The clause above will be satisfied by any truth assignment t unless $t(u_1)=F$, $t(u_3)=T$, and $t(u_8)=F$. A collection C of clauses over U is satisfiable iff there exists some truth assignment for U that simultaneously satisfies all the clauses in C. Such a truth assignment is called a *satisfying truth assignment.*

The formal statement of 3-Satisfiability given the above definitions is:

3-Satisfiability (3SAT)

INSTANCE: A set U of variables and a collection C of clauses over U.

QUESTION: Is there a satisfying truth assignment for C?

5.2 Time Complexity and the Importance of 3SAT

Given a function f and an algorithm A which computes result r we say that f is the time complexity function of A if the maximum number of steps A takes in order to compute r, given input of size n, is f(n). We also call f the *growth function* of A as f describes how the number of steps the algorithm executes grows with the size of the input. For example, the time complexity function for an algorithm that performs a top-to-bottom search of an array in order to find the largest element is f(n) = 3n+1. The algorithm must initialize the variable that will hold the largest value found so far (1 step), read in each of the n successive values (n steps), compare the current variable to the largest found so far (n steps), and switch them if necessary (maximum number of switches is n). The maximum number of steps in each phase of A are summed to find the maximum total number of steps taken by A. We call the actual number of steps taken by A to compute a given r the *time to solution.*

We say that a function f(n) is O(g(n)) whenever there exists a constant c such that $|f(n)| \leq c(|g(n)|)$ for all values of n greater than or equal to 0.([29]) If algorithm A has time complexity function f(n), and f(n) is O(g(n)), we will say A is O(g(n)), read "A is on the order of g(n)". It is possible for A to produce the correct result in fewer than f(n) steps on some or even most inputs. For A to be O(f(n)) it is only required that the algorithm take O(f(n)) steps for some non-empty set of inputs $I=\{i_1, i_2, ..., i_k\}$. We will call the case where the input to A is in I the *worst case.* The worst case for search algorithms is that in which the algorithm has to examine every point in the search space.

We often group algorithms together in time complexity classes based on their growth functions. For example, there is the class of algorithms whose growth functions are logarithmic. All algorithms in this class have growth functions that are O(log(n)). We say that a given problem *P* is in a complexity class *C* if the best known algorithm for solving *P* is in *C.* Although there is an infinite number of time complexity classes the two complexity classes P and NP contain the vast majority of all algorithms written

to solve practical problems. The techniques used to solve a problem in P are vastly different from the techniques used to solve a problem in NP.

The first class of problems contains those that are solvable in polynomial time, the class P. This class of problems is $O(n^k)$ where k is a constant. Problems in P are easy to solve, even for large values of n and k, because polynomials grow slowly. Examples of algorithms in P include the bubble sort ($O(n^2)$) and Floyd's algorithm to find the minimum distance between two vertices on a directed digraph ($O(n^3)$).([30]) The standard technique used to solve a problem in P is use of a deterministic algorithm. Problems in P usually yield readily to analysis and the resulting algorithm runs quickly and efficiently.

The second time complexity class we will consider contains those problems that are solvable in exponential time, the class NP. NP stands for non-deterministic polynomial time and is based on Turing machine theory. A non-deterministic Turing machine is allowed be in more than one state and is considered to have solved a problem if any of the states that it is in corresponds to a solution. An analogy can be made to a robot which is exploring a maze. A robot corresponding to a deterministic Turing machine can take only one road at each junction. A robot corresponding to a non-deterministic Turing machine can take every road at each junction. A problem which a non-deterministic Turing machine can solve in polynomial time is in NP. Unfortunately we cannot build non-deterministic Turing machines in practice so we have to settle for simulating them by deterministic machines. This raises the time complexity of the calculation from polynomial to exponential. Problems in NP are $O(c^n)$ with c a constant greater than 1. These problems are hard to solve even for small values of c and n because exponential functions grow very quickly. Even when c is only 2, moving from a problem instance of size n to one of size n+1 doubles the size of the search space. This doubles the number of steps a search algorithm uses in the worst case. Problems in NP are also known as *intractable* problems. Intractable problems make up most of the "interesting" problems faced today, with important intractable problems in graph theory (subgraph isomorphism [31]), scheduling (open-shop scheduling [32]), logic (satisfi-

ability [33]), automata theory (two-way finite state automaton non-emptiness [34]), and games (crossword puzzle construction [35]). The standard technique used to solve a problem in NP is to use a heuristic algorithm. Problems in NP are usually resistant to analysis with many different attempts to improve performance resulting in very little reduction in time to solution.

The subclass of NP containing the "hardest" problems in NP is known as the class of *NP-complete* problems. The NP-complete problems are the hardest in NP in the sense that any problem in NP can be transformed in polynomial time to an NP-complete problem. Therefore, if any polynomial time algorithm exists that can solve NP-complete problems, then that algorithm can be transformed in polynomial time to an algorithm that solves any problem in NP in polynomial time. No such algorithm is known to exist but there is no proof that such an algorithm cannot exist.

3SAT is an NP-complete problem. It is important not only as a challenging problem but also in what a heuristic used to solve 3SAT can tell us about heuristics used to solve other problems in NP. In the same way that any problem in NP can be transformed in polynomial time into an NP-complete problem, heuristics for one NP-complete problem p can often be adapted for use on another NP-complete problem p'. If there is no obvious way to adapt the heuristic it is always possible to transform p' into p in polynomial time and solve p' in that manner. This is the theoretical basis behind the target problem concept as introduced in section 2.3.3.

5.3 The Representation and Evaluation of Genes in the Genetic Encoding of 3SAT

Given the definitions in section 5.1 above and L=|U|, the representation of a truth assignment is straightforward. Each allele a_i takes values from A={0, 1}. The allele a_i contains the truth assignment for u_i where 0 represents an assignment of false and 1 represents an assignment of true.

We call the truth assignment which is the goal of the genetic search the *target truth assignment.* A group of clauses which is satisfied by the target truth assignment, the *target expression,* is randomly generated. Each gene is evaluated by the number of clauses in the test expression it satisfies. The number of clauses satisfied is used as the fitness score of the gene. Multiple satisfaction of the same clause counts as only one satisfaction of that clause. No fitness scaling is employed. The target truth assignment and the target expression are randomly generated before each run as is the initial population. The algorithm is terminated when the number of clauses satisfied is equal to the number of clauses contained in the target expression. The algorithm that generates the target expression is as follows and it guarantees that the target truth assignment satisfies the clauses.

A3: The algorithm to generate the target expression

1: A satisfying truth assignment is chosen.

2: For each clause in the target expression do

begin

The first literal is chosen at random from the target truth assignment;

The second and third literals are chosen at random;

end;

The number of clauses is set to 50L so that each variable appears as a literal, either negated or non-negated, in an average of 150 clauses. This is done in order to eliminate any satisfying truth assignments which are not the target truth assignment. As each new clause is added to the target expression the probability that a truth assignment that is not the target truth assignment might satisfy the target expression decreases because each additional clause is guaranteed to be satisfied by the target truth assignment but has only .875 probability of being satisfied by an alternative truth assignment. The existence of more than one satisfying truth assignment would complicate the analysis of results because the runs with target expressions that had multiple satisfying truth as-

signments would have a greater chance of finding a satisfying truth assignment than runs which had target expressions which were only satisfied by the target truth assignment, and it would not be immediately clear which runs had this property. The only way to check which runs had more than one satisfying truth assignment would be to exhaustively search all the possible truth assignments over the target clauses in each run. Setting the number of clauses to 50L is found to be empirically sufficient in preventing the occurrence of unwanted satisfying truth assignments. The population size n is set to 2L to ensure adequate variation in the initial population. An arbitrary upper value of 200 generations is set for each run. If the algorithm doesn't find the target truth assignment in 200 generations the program terminates.

5.4 Use of Bitwise Expected Value

Bitwise expected value is used in selected runs to choose the top n children in the temporary population T produced by overpopulation. The simple form of bev with no allele weighting is used as there was no analysis of the target expression before attempting to satisfy it.

It can be seen from the charts below that the runs in which bev is used show very similar performance to the runs in which the actual fitness function value is used. The average number of generations to solution is slightly smaller when the actual fitness function is used instead of the bev, as expected, but the importance of the results lies in the closeness between the average number of generations to solution for those runs in which bitwise expected value is used to estimate the fitness value and those runs in which the fitness function itself is used. The bev of a gene is easy to calculate, as has been demonstrated earlier, but the fitness function can be arbitrarily complex. Even though the average number of generations to solution is slightly higher in the bev case actual run time using bev is almost always less than run time using the actual fitness function. The more clauses that must be satisfied, the more time-efficient bitwise expected value becomes due to the increasing time each call to the fitness function takes.

Even though the difference in average number of generations to solution between algorithms which use bev and algorithms which use the fitness function grows with the number of variables this difference is overshadowed by the growing relative efficiency of bev in terms of calculation time when compared to the fitness function

5.5 Use of Overpopulation

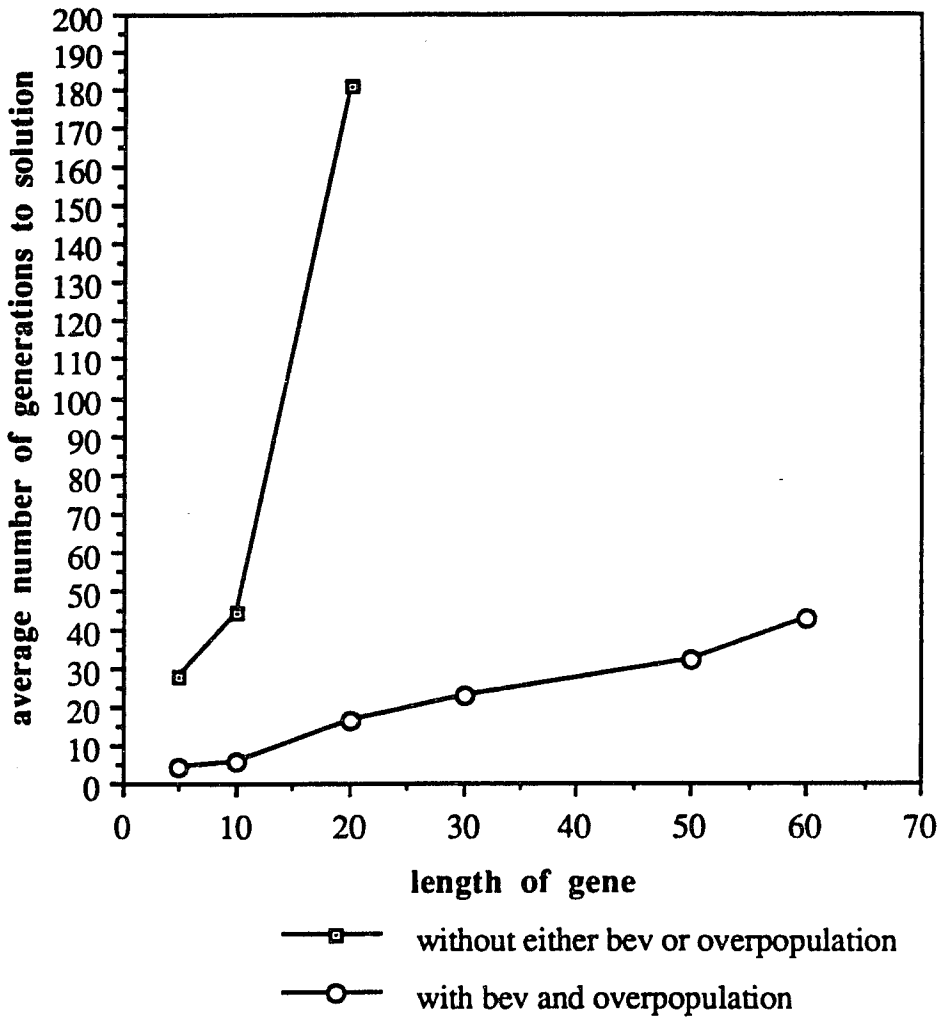
Overpopulation is used with M=2 for all runs. M is not varied during any run.

It is easy to see from the charts below that the runs in which overpopulation is used have much smaller average generations to solution than the runs in which overpopulation is not used. It is an order of magnitude difference, whether overpopulation is used with bitwise expected value or the actual fitness function. Overpopulation slows the growth rate of the average number of generations to solution down considerably, although it is of course still exponential. Without overpopulation the algorithm performs poorly, often tossing out good genes due to unlucky chance spins on the roulette wheel of selection. With overpopulation, even when the algorithm passes up good genes on the first n spins of the roulette wheel of selection, the algorithm shows a marked tendency to select good genes with the extra spins it is allotted to fill the temporary population T. This "second chance" effect makes a marked difference in the time to solution, as is evidenced in the graphs below.

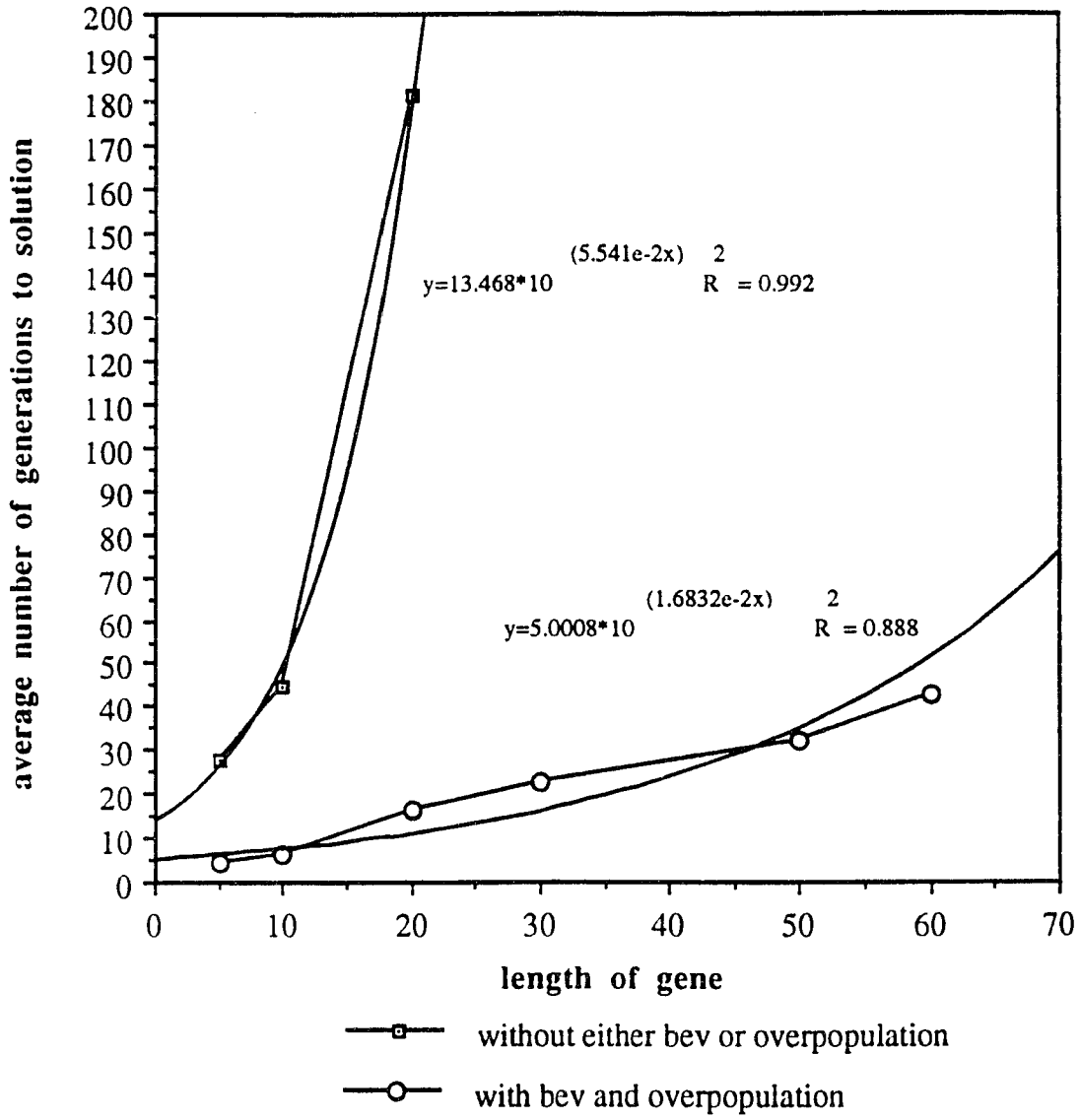
5.6 Results

The following graphs summarize the performance of the genetic algorithm on 3SAT. Three different experimental setups are used: the standard genetic algorithm, a genetic algorithm using overpopulation with bitwise expected value as the fitness estimator function, and a genetic algorithm using overpopulation with the actual fitness function as the fitness estimator function. Both algorithms which use overpopulation clearly outperform the algorithm which doesn't, both in average number of generations to solution and in wall-clock time to solution. Although the algorithm which uses bev

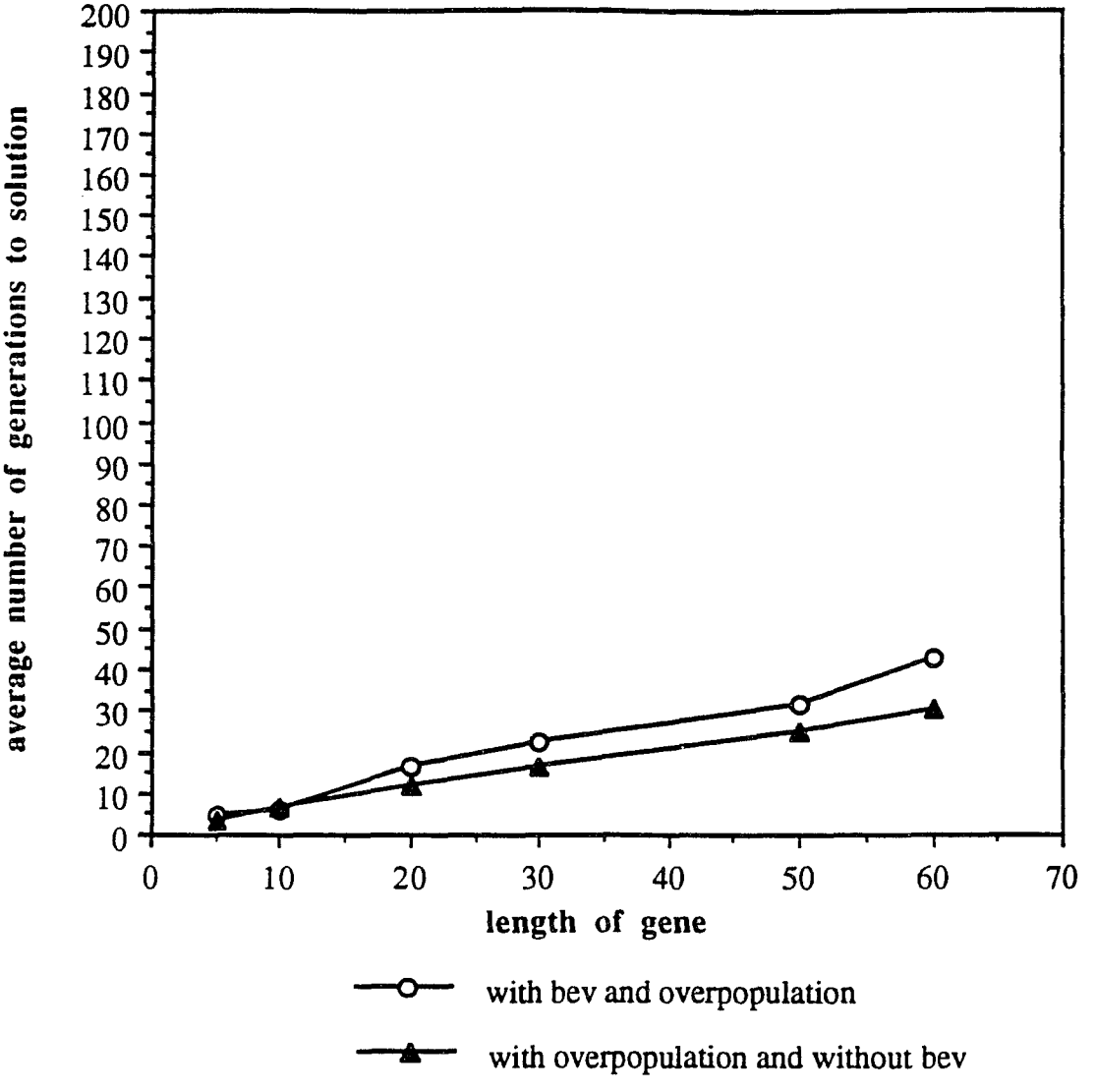
as the fitness estimator function takes more average generations to solution it takes less wall-clock time because bev is much more efficient to calculate. All graphs are also fitted with growth curves which estimate the average generations to solution for greater numbers of variables. It can be noted that the algorithm without overpopulation has a very steep growth curve, while the two algorithms with overpopulation show comparable growth curves. This suggests that the greater computational efficiency of bev will outweigh the smaller average generations to solution of the fitness function for even large numbers of variables.



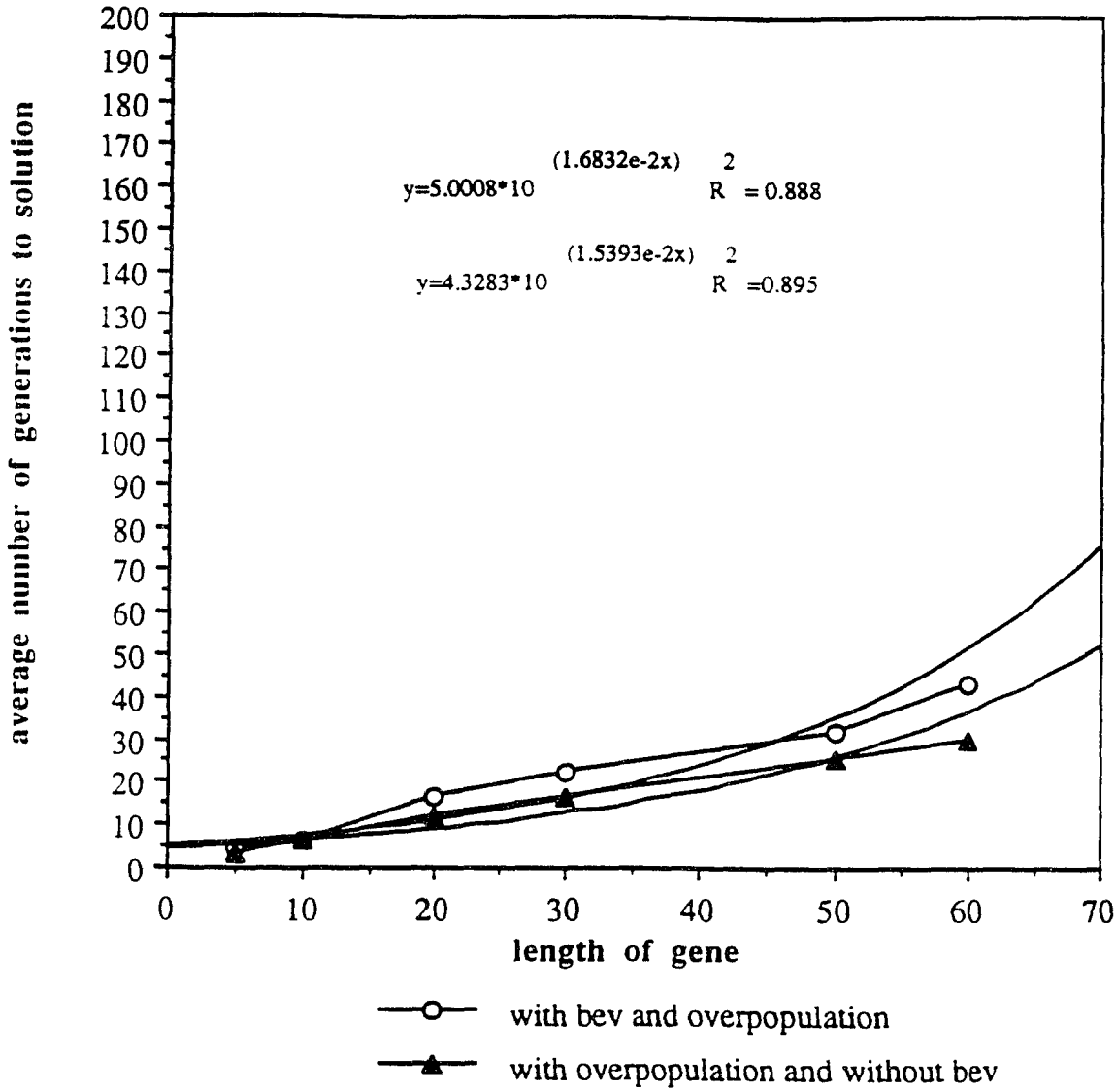
Graph 5.1 The SGA contrasted with a GA using overpopulation and bev



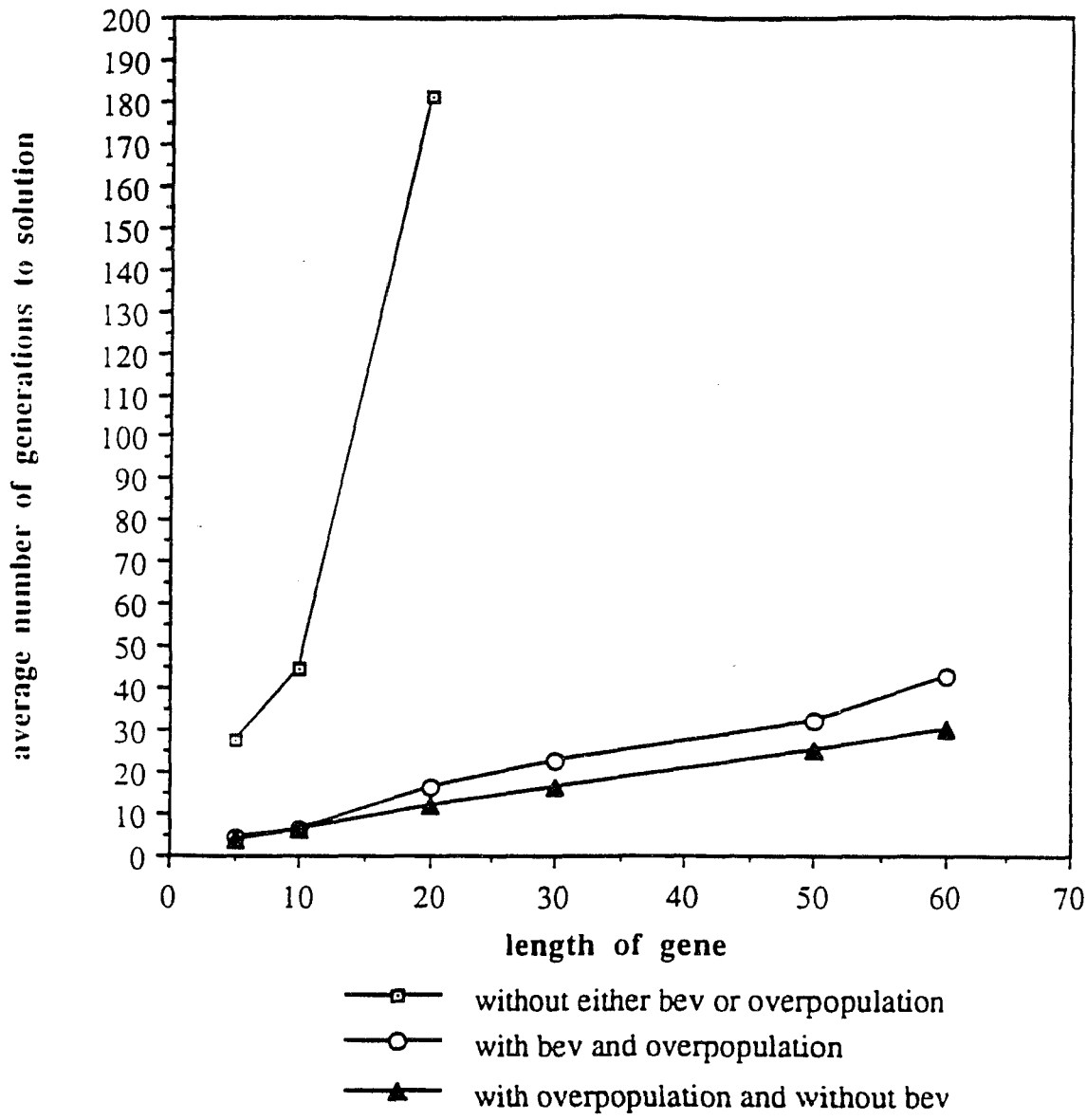
Graph 5.2 The SGA contrasted with a GA using overpopulation and bev - rate of growth



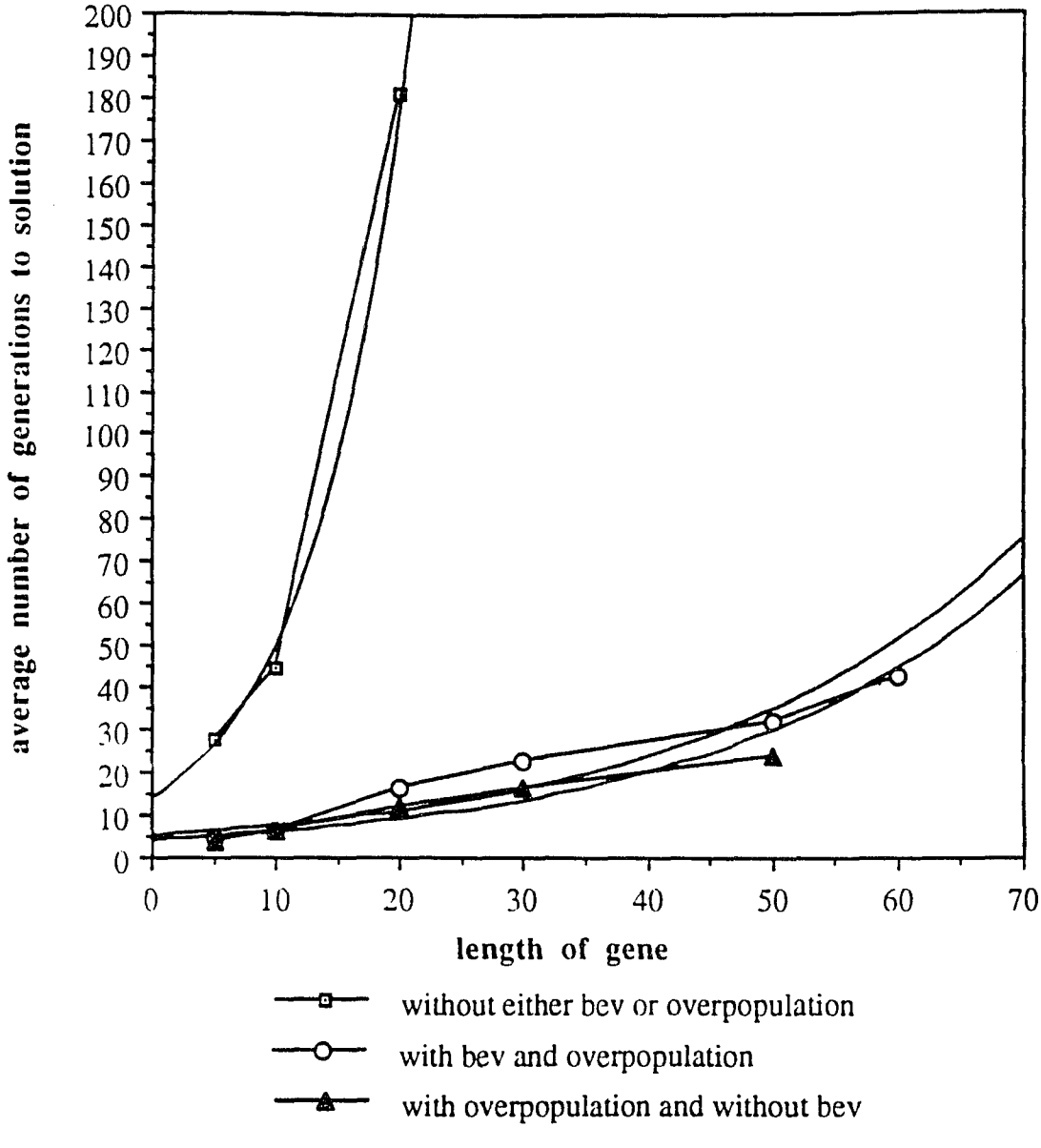
Graph 5.3 A GA using overpopulation and bev contrasted with a GA using overpopulation and the fitness function



Graph 5.4 A GA using overpopulation and bev contrasted with a GA using overpopulation and the fitness function - rate of growth



Graph 5.5 A comparison of the three algorithms



Graph 5.6 A comparison of the three algorithms - rate of growth

6. Transforming the TSP to SAT

As an example of the transformation of a problem instance into a target problem we have chosen the Traveling Salesman Problem (TSP). The transformation in this case will be from an instance of the traveling salesman problem and an upper bound on the length of the allowable tour(s) to a boolean expression. Because both the TSP and SAT are NP-complete, we know such a transformation exists. In fact, we know such a transformation can be computed in polynomial time ([47]). The TSP is the subject of intense study in the field of combinatorial optimization ([19][46]) so it serves as an excellent example of the target problem concept. Overpopulation and bitwise expected value will be used in the genetic algorithms that attempts to satisfy the boolean expression that results from the transformation of the TSP instance.

6.1 The Traveling Salesman Problem and its Importance

The Traveling salesman problem is easy to state intuitively. You are a traveling salesman who is given an airplane and assigned the job of visiting each city in your district. You have to pay for jet fuel out of your own pocket, so you want to use as little as possible. Sitting Down with a map which has marked upon it the air mileage from every city in your district to every other city in your district, you plan out a course. There are some obvious guidelines to follow. Visiting a city more than once is a waste of gas, and so is flying back and forth across the district visiting distant cities before near ones. After some thought a tour emerges where you visit each city exactly once and return to your starting point. It occurs to you, however, that there might be a shorter route which accomplishes the same objective. Is there?

There are many different, equivalent formal ways of stating the traveling salesman problem. We follow ([48]) in our formal definition, and then give an equivalent statement.

Traveling Salesman Problem (TSP)

INSTANCE: A finite set $C=\{c_1, c_2, ..., c_m\}$ of "cities", a "distance" $d(c_i, c_j)$ for each pair of cities c_i and c_j in C, and a positive integer β called the *bound.*

QUESTION: Is there a "tour" of all the cities in C having total length no more than β, that is, an ordering $<c_{\pi(1)}, c_{\pi(2)}, ..., c_{\pi(m)}>$ of C such that

$$\left[\sum_{i=1}^{m-1} d(c_{\pi(i)}, c_{\pi(i+1)})\right] + d(c_{\pi(m)}, c_{\pi(1)}) \leq \beta$$

Our goal is to take an instance of the TSP T and a bound β and transform T into a boolean expression B such that the only satisfying truth assignments for B are those that correspond to tours of T which have mileage less than or equal to β.

An alternative definition of the TSP is as a problem on a weighted, complete, undirected graph G=(V, E). We will call the vertices cities, with C=|V|. We will call the edges *roads*, and road k will be denoted by r_k. R=|E| is the number of roads. If $r_k=(c_i, c_j)$ then $length(r_k)=d(c_i, c_j)$. We say r_k enters c_i if $r_k=(c_i, c_j)$ for some c_j. We define the mileage of a tour t (denoted mileage(t)) as the sum of the lengths of all the roads on t.

6.2 Notation

The + symbol will stand for logical or, concatenation of literals and clauses will stand for logical and, the - symbol will stand for logical negation, and the → symbol will stand for logical implication.

We will describe B as being made up of subexpressions b_1, b_2, b_3 and b_4, the conjunction of which will form B. This is strictly for the convenience of the reader. We could, if we wanted, uniquely rename the variables in b_1, b_2, b_3 and b_4 and combine them within a single boolean expression B′. Note that it is only necessary for the genetic algorithm to represent those portions of the tour that allow us to calculate in poly-

nomial time the truth assignment corresponding to that tour. In particular, in subexpression b_4 we calculate the length of the tour that gene γ represents. We can do this in polynomial time via a lookup table and summation of road lengths. The mileage of the tour that γ represents is uniquely determined by γ, but is not explicitly part of γ in that no allele(s) in γ represents the mileage of the tour that γ represents.

In b_1, b_2 and b_3 the boolean variables correspond directly to roads, and we will construct these expressions using the roads r_1, r_2, ..., r_R as if they were boolean variables. Our truth assignment for these sections is T for those roads the salesman travels on a trial solution and F for all other roads.

6.3 Visiting Each City at Least Once

In this section we define b_1.

In the TSP each city must be visited at least once and each road can be used at most once. If we imagine the salesman traveling on his route, we can see that for each city the salesman must enter on one road and leave on another. We can enforce this limitation for each city c_i by forming a clause δ_i for c_i in the following manner.

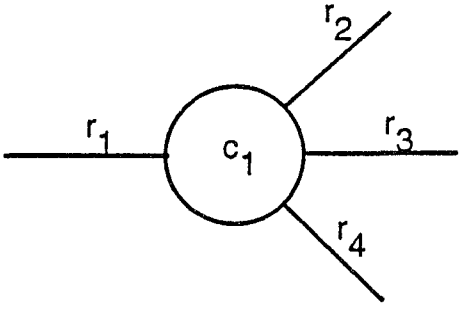


Fig. 6.1 Example city

Roads r_1, r_2, r_3 and r_4 run through city c_1 in Fig. 6.1, above. The clause δ_1 that ensures that city c_1 will be visited at least once will be the disjunction of all conjunctions of roads $r_j r_k$ such that r_j and r_k enter c_i and $k \neq j$. For c_1, $\delta_1=(r_1r_2+r_1r_3+r_1r_4+r_2r_3+r_2r_4+r_3r_4)$. Because the route our salesman is taking is undirected (it

doesn't matter in which direction the salesman makes his journey, only the total distance traveled) the conjunction of roads $r_i r_j$ is equivalent to $r_j r_i$. A clause δ_i is formed for each c_i. The subexpression b_1 is the conjunction of all δ_i.

$$b_1 = \bigwedge_{i=1}^{C} \delta_i$$

6.4 Visiting Each City Exactly Once

In this section we define b_2.

The subexpression b_2 ensures that our salesman visits no city more than once. Since he is restricted to visit each city at least once by b_1 (defined in section 6.3 above) the conjunction $b_1 b_2$ will ensure that each city is visited exactly once.

As our salesman visits a city c_i he enters and leaves c_i by different roads. Once he has visited c_i for the first time he can never travel a pair of roads that would lead him through c_i again. Using our previous example from Fig. 6.1 in section 6.3, once our salesman has passed through c_1 on road pair $r_1 r_2$, he can no longer take any of the road pairs $r_1 r_3$ or $r_1 r_4$ or $r_2 r_3$ or $r_2 r_4$ or $r_3 r_4$, because using any of these road pairs would lead him through c_1 again. In the notation of this section, we need to form the expression θ'_1 =

$$(r_1 r_2 \rightarrow \text{-}(r_1 r_3))(r_1 r_2 \rightarrow \text{-}(r_1 r_4))(r_1 r_2 \rightarrow \text{-}(r_2 r_3))(r_1 r_2 \rightarrow \text{-}(r_2 r_4))(r_1 r_2 \rightarrow \text{-}(r_3 r_4))$$

$$(r_1 r_3 \rightarrow \text{-}(r_1 r_2))(r_1 r_3 \rightarrow \text{-}(r_1 r_4))(r_1 r_3 \rightarrow \text{-}(r_2 r_3))(r_1 r_3 \rightarrow \text{-}(r_2 r_4))(r_1 r_3 \rightarrow \text{-}(r_3 r_4))$$

$$(r_1 r_4 \rightarrow \text{-}(r_1 r_2))(r_1 r_4 \rightarrow \text{-}(r_1 r_3))(r_1 r_4 \rightarrow \text{-}(r_2 r_3))(r_1 r_4 \rightarrow \text{-}(r_2 r_4))(r_1 r_4 \rightarrow \text{-}(r_3 r_4))$$

$$(r_2 r_3 \rightarrow \text{-}(r_1 r_2))(r_2 r_3 \rightarrow \text{-}(r_1 r_3))(r_2 r_3 \rightarrow \text{-}(r_1 r_4))(r_2 r_3 \rightarrow \text{-}(r_2 r_4))(r_2 r_3 \rightarrow \text{-}(r_3 r_4))$$

$$(r_2 r_4 \rightarrow \text{-}(r_1 r_2))(r_2 r_4 \rightarrow \text{-}(r_1 r_3))(r_2 r_4 \rightarrow \text{-}(r_1 r_4))(r_2 r_4 \rightarrow \text{-}(r_2 r_3))(r_2 r_4 \rightarrow \text{-}(r_3 r_4))$$

$$(r_3 r_4 \rightarrow \text{-}(r_1 r_2))(r_3 r_4 \rightarrow \text{-}(r_1 r_3))(r_3 r_4 \rightarrow \text{-}(r_1 r_4))(r_3 r_4 \rightarrow \text{-}(r_2 r_3))(r_3 r_4 \rightarrow \text{-}(r_2 r_4))$$

Using DeMorgan's laws and boolean algebra we can reduce θ'_1 to the expression θ_1

$$(\text{-}r_1 + \text{-}r_2 + \text{-}r_3)(\text{-}r_1 + \text{-}r_2 + \text{-}r_4)(\text{-}r_1 + \text{-}r_3 + \text{-}r_4)(\text{-}r_2 + \text{-}r_3 + \text{-}r_4)(\text{-}r_1 + \text{-}r_2 + \text{-}r_3 + \text{-}r_4)$$

Since each of the original clauses contained two pairs of roads and we know there are R^2 pair of roads (by definition of R) we have at most $(R^2)^2$, or R^4, clauses in θ'_i for all i. This satisfies the requirement that our transformation be polynomial time, as we can write a polynomial-bounded number of clauses in polynomial-bounded time. The simplification of θ'_1 to just the 5 clauses in θ_1 shows that the resulting expression θ_i is likely to be much smaller than θ'_i. The expression θ_i will contain every possible disjunction of 3 roads that lead into c_i plus every possible disjunction of 4 roads that lead into c_i. As every possible disjunction of 4 roads will be subsumed by some combination of disjunctions of 3 roads, we need only count the number of disjunctions of 3 roads. As C-1 roads lead into c_i (one road from each city other than c_i) θ_i will thus have exactly

$$\binom{C-1}{3}$$

clauses, which is equal to

$$\frac{(C-1)(C-2)(C-3)}{6}$$

clauses when C>3. Form θ_i for each c_i. The conjunction over all θ_i is b_2.

$$b_2 = \bigwedge_{i=1}^{C} \theta_i$$

Let *P* be the set of all permutations of R or fewer roads. Let Q be the subset of *P* which contains only those permutations of roads which have the property that every city is on a cycle. For a permutation *p* to be in Q each city in *p* must be of degree two. In the terminology of this section each city must be visited by the salesman at least once, but no more than once, entering and leaving on different roads.

The expression b_1 ensures that each city is visited exactly once, and that the salesman enters on one road and leaves on another. The expression b_2 ensures that each city is visited no more than once. As these are the two requirements for a permutation of

roads to be a member of Q we can conclude that any truth assignment that satisfies b_1b_2 represents a permutation of roads in which every city lies on a cycle.

6.5 Ensuring Connectivity

This does not, however, ensure that a truth assignment that satisfies b_1b_2 represents a tour. For a permutation of roads p in Q to be a tour it is necessary that p form only one cycle, a cycle that includes all cities. The subexpressions b_3 is satisfied only by trial solutions with this property.

6.5.1 Solving the problem of non-connected subgraphs

In this section we define b_3, using roads as the input variables and cities as internal variables.

The expression b_1b_2 ensures that each city is on a cycle but is satisfies by truth assignments which represent solutions with disjoint subcycles. A legal solution to the TSP, however, contains only one cycle, the complete tour. The previous expressions b_1 and b_2 must be further augmented with an expression that is only satisfied by those truth assignments corresponding to single-cycle subgraphs (tours).

For each city c_i let $X_i=(c_{i1}, c_{i2}, ..., c_{iC}\}$ be the set of all cities that share a road with c_i and let $E_i=\{r_{i1}, r_{i2}, ..., r_{iC}\}$ be the set of roads that connect c_i to cities $c_1, c_2, c_3, ...,$ c_C, respectively. with the exception that c_i does not share a road with c_i and there is no road connecting c_i to c_i. We will continue the notation of the previous section in the use of roads as variables, and broaden that usage to include cities.

For each city c_i form the clause η_i

$$\eta_i=c_i\text{->}((c_{i1}r_{i1})+(c_{i2}r_{i2})+...+(c_{iC}r_{iC}))$$

where $c_{ir}\varepsilon X_i$ and $r_{is}\varepsilon E_i$. Each η_i with $0<i<C+1$ states the following fact: to reach c_i the salesman must have traveled along a road r_{is} that connects a city c_{ir} which he has already visited to c_i. We must also form these two additional clauses $\eta_0=c_1$ ($\eta_0=T\text{->}c_1$,

where T stands for the boolean constant true) and the clause $\eta_{C+1}=(c_1c_2 ...c_C)$. The clause η_0 states that he starts his trip in c_0 and the clause η_{C+1} states that he must visit all cities in his district. The expression b_3 is the conjunction of all η_i.

$$b_3 = \bigwedge_{i=0}^{C+1} \eta_i$$

The most intuitive way to understand b_3 is to view the salesman as an automated theorem prover. He is given the goal of satisfying η_{C+1} given the inference rules η_1 through η_C and the single axiom η_0. He has two sets of cities, those which he has visited, V, and those which he has not visited, NV. To satisfy η_{C+1} he must satisfy the right-hand side of each η_i. Given a subexpression η_j, we can add c_j to V if there is a conjunction $c_k r_m$ in the right hand side of η_j where c_k is in V and r_m is true in this trial solution. At the beginning of this process h_0 puts c_1 in V. All other cities are in NV. The salesman continues with this process until no other roads can be added to V. He then checks to see if every city is in V, which corresponds to satisfying η_{C+1}. If all cities are in V then he has visited every city and satisfied b_3.

6.6 Finishing With Total Mileage Less Than β

Any truth assignment that satisfies $b_1b_2b_3$ represents a tour of T. In fact with $b_1b_2b_3$ we have transformed the Hamiltonian Circuit problem into SAT, another example of the power of the target problem concept. To finish our transformation we must ensure that no tour satisfies B which does not also have length less than or equal to β.

6.6.1 The problem of mileage

There is no obvious way to incorporate mileage into a boolean expression. Boolean variables can take on only two values, either true of false, while mileage can be any positive integer.

The solution lies in base 1 notation for the positive integers, also known as the prisoner's tally. A number in unary notation consists of a sequence of marks whose length is equal to the size of the number to be represented. For example, the number 5 is written in unary notation as 11111 and the number 10 is written as 1111111111. We will use a slight modification of the standard unary notation in that we will allow 0's to be written to the left of a number in unary notation without changing the value of that number. So, for example, 5=11111=011111=0011111 and so on. We will call the number of 1's plus the number of 0's in a number in modified unary notation the *total length* of that number. Given any two positive integers x and y with x>y, we can pad the modified unary representation of y to have total length x by adding x-y 0's to the left end of the modified unary representation of y.

The modified unary notation we defined above has obvious parallels to the notation for a truth assignment that we used in solving 3SAT problems. When we use the obvious mapping 0=F and 1=T we can view numbers in our modified unary notation as truth assignments, and we can write boolean expressions that these truth assignments either satisfy or fail to satisfy. We say a number N maps to a truth assignment N when, in modified unary notation and with 0=F and 1=T, N=N. With this mapping, we will say that a number N satisfies a boolean expression b iff N maps to N and N satisfies b.

Let $\Omega=\Sigma\text{length}(r_i)$. We know that Ω is greater than the length of any allowable tour because every allowable tour of a fully connected graph leaves out at least one road. It is easy to see that we can uniquely pad the modified unary representation of mileage(t) to have total length Ω, as mileage(t)<Ω.

Let $<v_\Omega, v_{\Omega-1}, ..., v_1>$ be a sequence of boolean variables. We write mileage(t) in unary notation as *t* and map *t* to the truth assignment τ with Ω variables (padding if necessary) and notice that τ assigns true to all v_i with i<mileage(t)+1. In particular, the positive integer β when mapped to a truth assignment in the manner described above assigns true to all v_i where i<β+1.

With this knowledge we can conclude that any tour t for which mileage(t)>β will assign $v_{\beta+1}$ a value of true, and any tour t′ for which mileage(t′)≤β will assign $v_{\beta+1}$ a value of false. Therefore

$$b_4=-v_{\beta+1}$$

All that remains is, when given a tour t, to observe whether mileage(t) satisfies b_4.

This concludes our transformation.

6.7 Results

This section contains the results of transforming TSPs into boolean expressions using the method described above. Fitness scaling was employed. The result of subexpression b_1 is multiplied by 2. This is an empirical decision based on analysis of results. The result of subexpression b_2 is scaled so that, if the number of roads n_r entering a city c is greater than 2 (thus falsifying b_2) the gene is penalized n_r-2 fitness points. This is also an empirical decision. The value of b_3 for a given gene is the average value of b_3 when each position is the starting point. This keeps the first gene from exerting undue influence over the algorithm. Subexpression b_3 is only evaluated if b_1 and b_2 are both true. Subexpression b_4 returns the following number i

A4: The algorithm that computes the value returned by b_4

1: i←1.

2: d←0.

3: While i≤the length of the gene and d<β do

begin

if the allele at position i has value 1

then d←d+the length of road i;

i←i+1;

end;

This is an empirical decision. There is very little selective pressure to find a tour with length less than β as long as the reward for doing so is the satisfaction of only one clause.

6.7.1 Empirical results

The algorithm is tested on a small (10 city) TSP, a description of which is given in the appendices. The chromosome length is therefore 45, one allele for each road. The crossover rate is set at 0.6, and the mutation rate is 0.02. A limit of 600 generations is set for the algorithm. Overpopulation is used with M=2 and bitwise expected value is used as the fitness estimation function. Each data point corresponds to the average value for 20 runs.

The independent variable β is the maximum allowable tour. That is, if the algorithm finds a tour with length less than or equal to β it is considered to have solved that instance and execution halts. The algorithm shows exponential growth in average generations to solution as β more closely approximates the optimum tour length. There is reason to believe, however, that a good estimate of optimum tour length can be obtained without having β get too close to the optimum. This will be explained in the next section.

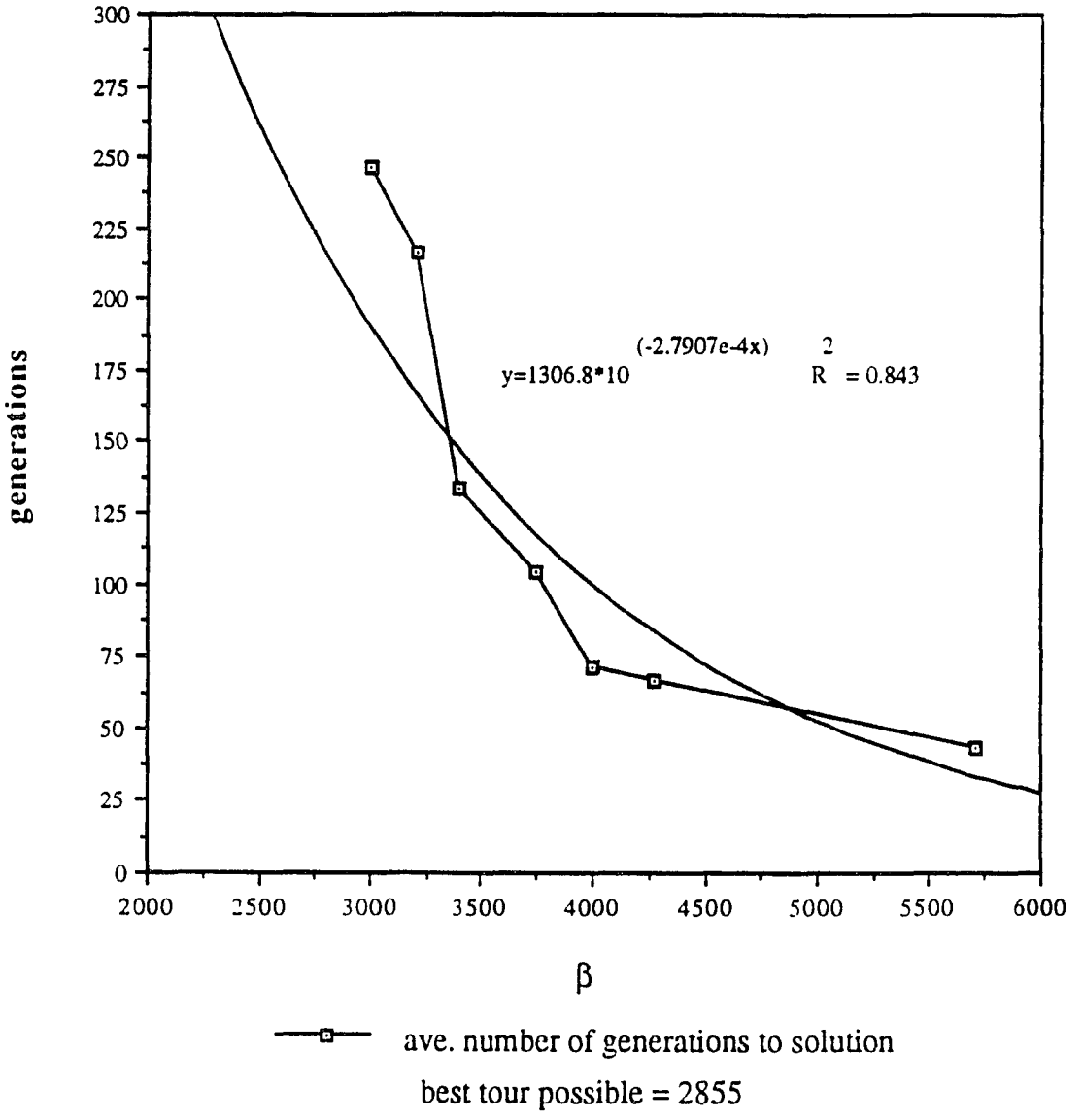
The algorithm does very well in finding an acceptable tour for all values of β. It finds an acceptable tour on every run in which β is greater than the optimum, and on 40% (8 of 20) runs in which β is the optimum.

6.7.2 Some interesting convergence properties

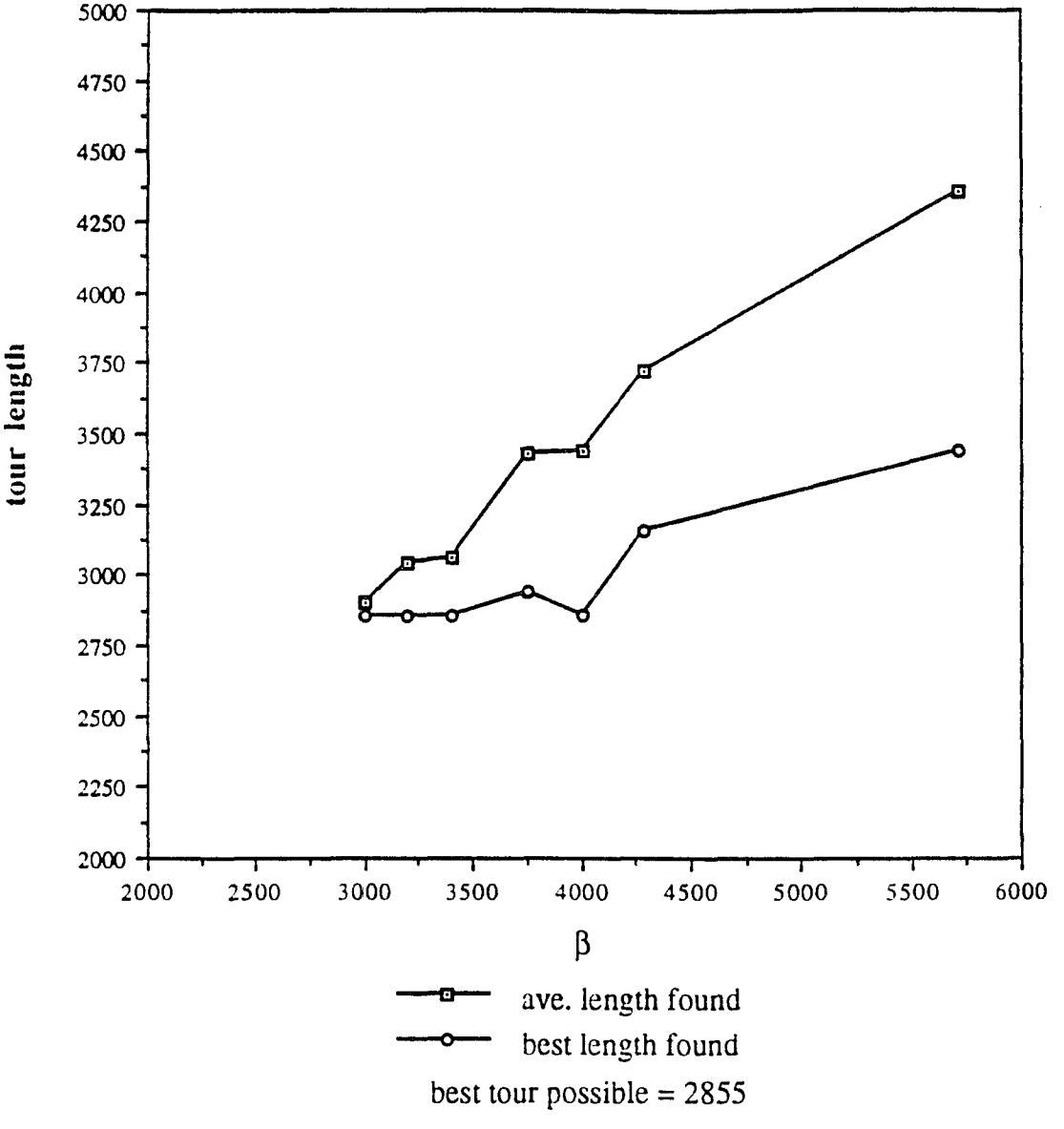
When the best value found over 20 runs at each β is plotted against the average value found over the same 20 runs an interesting pattern emerges. A logarithmic curve fitted to the average data and an exponential curve fitted to the best data intersect at a point on the y axis close to the optimum tour mileage. Graph 6.4 shows this process as applied to only those runs in which the algorithm succeeded in finding a satisfactory tour 100% of the time. An even better estimate can be provided by also using data from

runs which did not find a satisfactory tour every time. This is shown in Graph 6.5. Extrapolating in this manner, it might be expected that several relatively fast runs with large values of β might serve as a basis for the estimation of the length of the optimal tour.

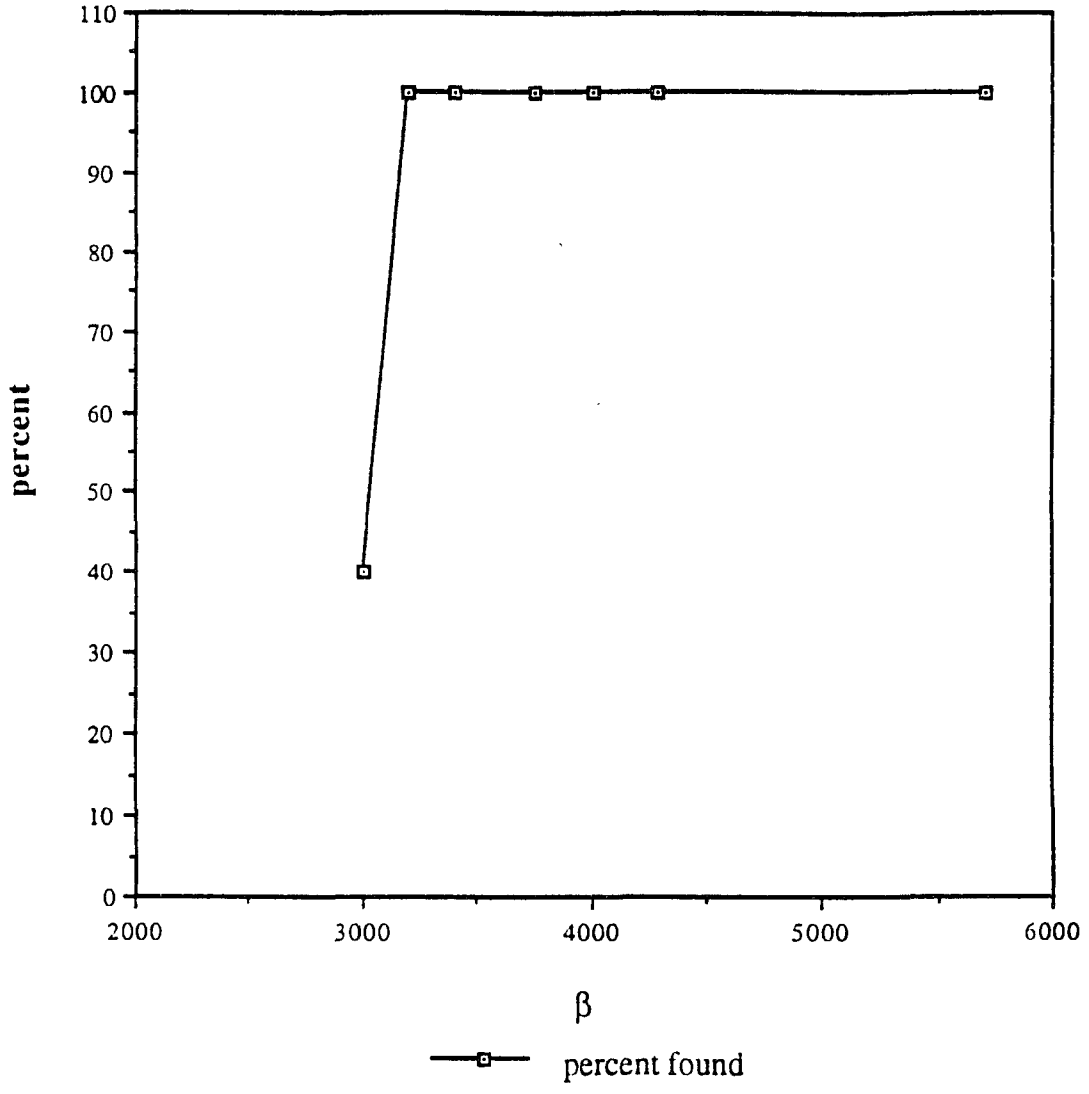
The value of the average tour is bounded above by a horizontal line at the mileage corresponding to the longest tour so it is reasonable to suppose that it follows a logarithmic growth function. It also seems reasonable to assume that the best tour found would grow exponentially as β grew ([49]) because there are many more longer tours than there are shorter ones. The estimate does not change very much if it is assumed that the best tour found grows only linearly with β.



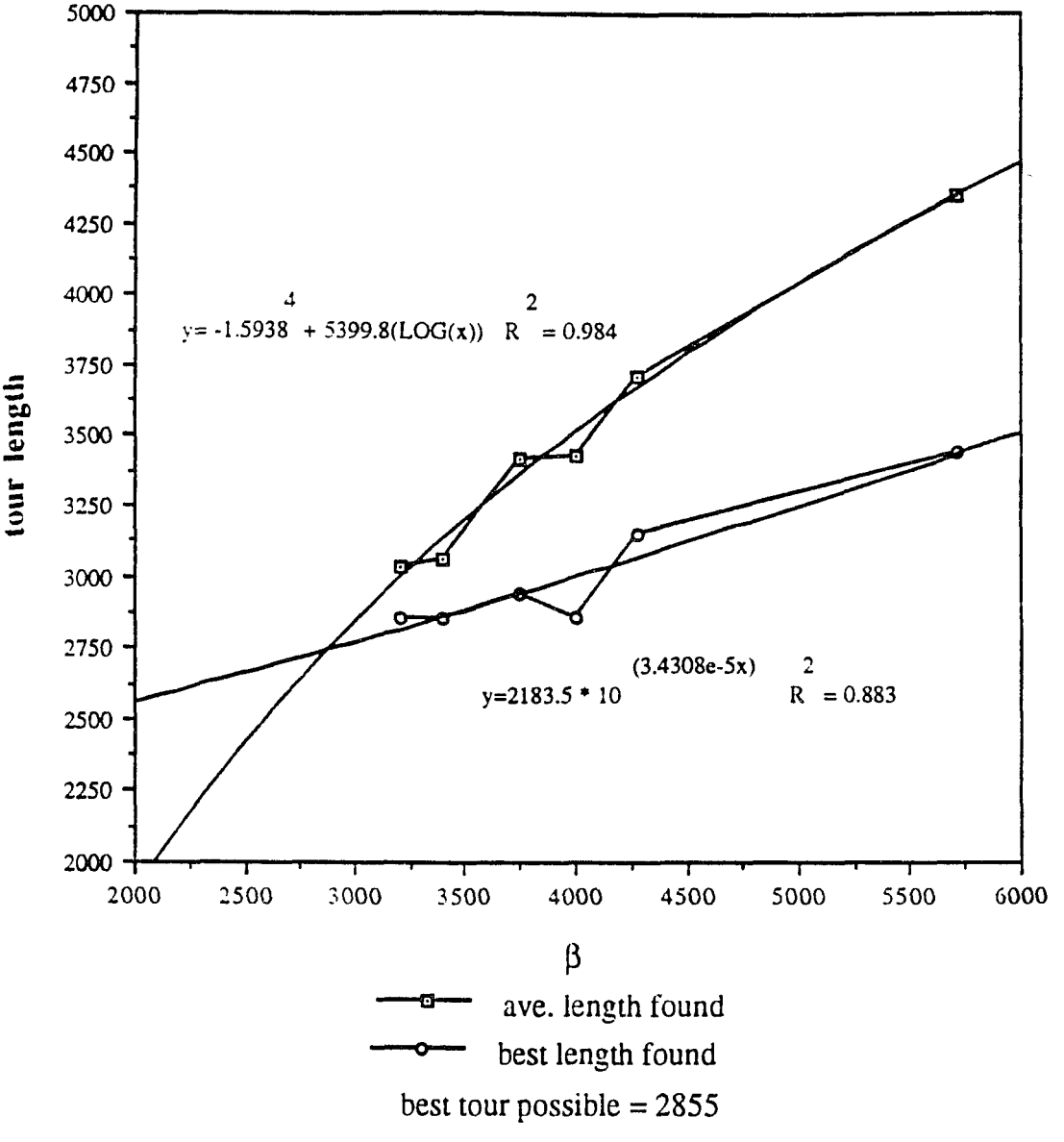
Graph 6.1 Average generations to solution



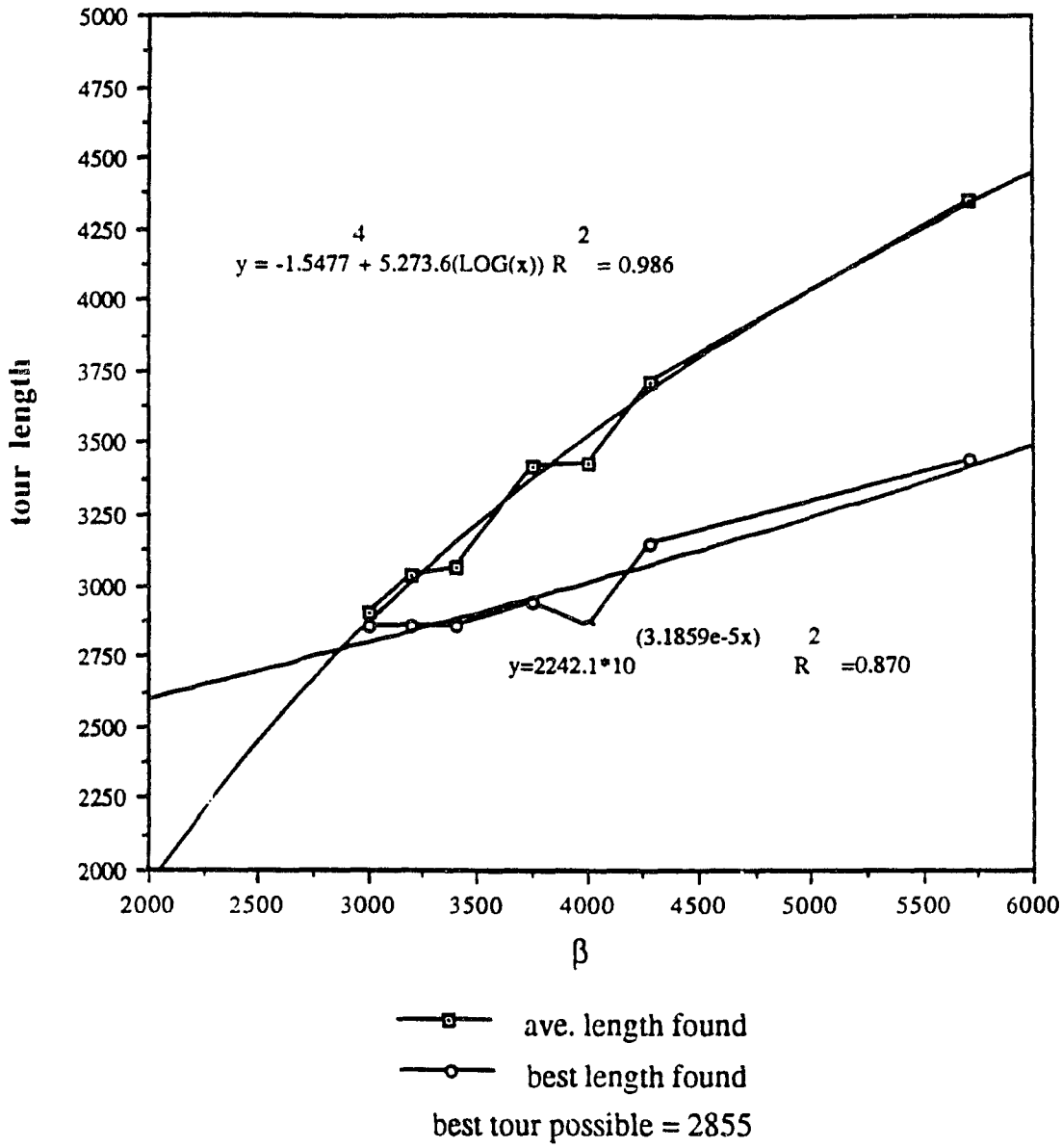
Graph 6.2 Average and best solutions found



Graph 6.3 Percentage of tours found within 600 generations



Graph 6.4 Estimation of minimum tour



Graph 6.5 Better estimate of minimum tour

7. Summary

We have introduced two new genetic methods: bitwise expected value and overpopulation. The introduction of overpopulation involves the generalization of the schema growth equation in order to reflect the new manner in which schema are added to the population. This generalization shows that a genetic algorithm using overpopulation will experience more rapid growth of above average schema than a genetic algorithm without overpopulation. If the building block hypothesis is correct, faster growth of above average schema will lead to a reduction in the number of individuals which must be evaluated in order to solve a problem. Bitwise expected value (bev) is introduced as a fitness estimation function. The genes in the temporary population set up be overpopulation are evaluated by bitwise expected value and the best are kept as the next generation. Bev is justified by analogy with the schema theorem. Good genes are made up of good schema, and a genes fitness is a function of the schema it contains. Bitwise expected value works from top to bottom. Good genes are good genes because their individual alleles are good, so alleles in good strings have high expected value in a gene's descendants. This analogy suggests that the bev of a gene is likely to be very close to its fitness value.

Empirical evidence is presented to test the theoretical claims of faster above average schema growth using overpopulation and accuracy of fitness value prediction by bitwise expected value. The testbed problem is 3SAT, an NP-complete problem. Excellent results were found, with the combination of overpopulation and bev outperforming the standard genetic algorithm by an order of magnitude. This clearly suggests that overpopulation is instrumental in the production and propagation of above average schema. The correlation between bitwise expected value and the actual fitness function evaluation of a gene is measured and found to be high, significant at well over the $p<0.0001$ level for every run.

The target problem concept is introduced as a way of avoiding problems in representation design. In the target problem concept the problem to be solved by a genetic algorithm is transformed into an instance of a problem for which a suitable representation is already known. The target problem concept is then applied to the Traveling Salesman Problem, transforming it into a boolean expression to be satisfied. The same methods are applied to this expression as to the previous, "pure", satisfiability problems with excellent results. The length of the acceptable tour is gradually decreased from run to run until the shortest tour is found. A curious convergence property is noted whereby runs in which relatively long tours are allowed are good estimators of the length of the optimal tour.

Several extensions to overpopulation and bitwise expected value are discussed, including dynamic choice of the overpopulation size parameter M and differential weighting of alleles in the calculation of bev.

References

1. Schaffer, J. D. ed., *Genetic Algorithms: Proceedings of the Third International Conference*, Morgan Kaufmann, 1989

2. Belew, Richard K. ed., *Genetic Algorithms: Proceedings of the Fourth International Conference*, Morgan Kaufmann, 1991

3. Porter, B. W. ed., *Proceedings of the Seventh International Conference on Machine Learning*, Morgan Kaufmann, 1990

4. Rawlins, G. ed., *Foundations of Genetic Algorithms*, Morgan Kaufmann, 1990

5. Holland, J., *Adaptation in Natural and Artificial Systems*, Ann Arbor: The University of Michigan Press, 1975

6. Goldberg, D. *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley, Reading, Massachusetts, 1989, pp. 36-38

7. ibid., pp 29-30

8. Valiant, L. "A Theory of the Learnable", in *Communications of the ACM*, vol. 27, no. 11, Nov. 1984, pp. 87-106

9. Angluin, D., "Learning Regular Sets from Queries and Counterexamples", in *Information and Computation 75*, 1987, pp. 87-106

10. Kaelbling, L., "Learning Functions in k-DNF from Reinforcement", in *Proceedings of the Seventh International Conference on Machine Learning*, 1990, pp. 162-169

11. DeJong, K. A., and Spears, R., "Using Genetic Algorithms to Solve NP-Complete Problems", in *Genetic Algorithms: Proceedings of the Third International Conference, 1989*

12. McCallum, R., and Spackman, K., "Using Genetic Algorithms to Learn Disjunctive Rules from Examples", in *Proceedings of the Seventh International Conference on Machine Learning, 1990*

13. Bonelli, P., et al, "NEWBOOLE: A Fast GBML System", in *Proceedings of the Seventh International Conference on Machine Learning, 1990*

14. Gu, J., "Efficient Local Search for Very Large-Scale Satisfiability Problems", in *SIGART Bulletin*, vol. 3, no. 1, pp.8-12

15. Goldberg, A., et al, "Average Time Ananlysis of Simplified Putnam-Davis Procedures", in *Information Processing Letters*, vol. 15, no. 2, 1982, pp. 72-75

16. Chvatal, V., and Szemeredi, E., "Many Hard Examples for Resolution", in *Journal of the ACM*, vol. 35, no. 4, Oct. 1988, pp.759-768

16. Chvatal, V., and Szemeredi, E., "Many Hard Examples for Resolution", in *Journal of the ACM*, vol. 35, no. 4, Oct. 1988, pp.759-768

17. Foulds, L. R., *Combinatorial Optimization for Undergraduates*, Springer-Verlag. 1984, pp. 170-177

18. Cooper, L., and Steinberg, D., *Introduction to Methods of Optimization*, W. B. Saunders Co., 1970, pp. 325-327

19. Lawler, E. L., et al, *The Traveling Salesman Problem*, John Wiley and Sons, New York, 1985

20. Colorni, A., et al, "Genetic Algorithms and Highly Constrained Problems: The Time-Table Case", in *Parallel Problem Solving from Nature (PPSN 1)*, Springer-Verlag, 1991, pp. 55-59

21. Husbands, P., et al, "Genetic Algorithms, Production Plan Optimisation and Scheduling", in *Parallel Problem Solving from Nature (PPSN 1)*, 1991, pp. 80-84

22. Ulder, N., et al, "Genetic Local Search Algorithms for the Traveling Salesman Problem", in *Parallel Problem Solving from Nature (PPSN 1)*, 1991, pp. 109-116

23. Braun, H., "On Solving Traveling Salesman Problems by Genetic Algorithms", in *Parallel Problem Solving from Nature (PPSN 1)*, 1991, pp. 129-133

24. Kabad, N., and Nygard, K., "Improving the Performance of Genetic Algorithms in Automated Discovery of Parameters", in *Proceedings of the Seventh International Conference on Machine Learning, 1990*, pp. 140-148

25. DeGaris, H., "Genetic Programming: Building Artificial Nervous Systems Using Genetically Programmed Neural Network Modules", in *Proceedings of the Seventh International Conference on Machine Learning, 1990*, pp. 132-138

26. Suh, J., and Lee, C., "Operator-Oriented Genetic Algorithm and Its Application to the Sliding Block Puzzle Problem", in *Parallel Problem Solving from Nature (PPSN 1)*, 1991, pp. 98-103

27. Liepins, G., and Wang, L., "Classifier System Learning of Boolean Concepts", in *Genetic Algorithms: Proceedings of the Fourth International Conference, 1991*, pp. 318-323

28. Garey, M., and Johnson, D., *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Co., New York, 1979, pp. 38-39, 46

29. ibid., p. 6

30. Even, S., Graph Algorithms, Computer Science Press, Rockville, Maryland, 1979

31. Garey, M., and Johnson, D., *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Co., New York, 1979, p. 202

32. ibid., p. 241

33. ibid., p. 259

34. ibid., p. 265

35. ibid. p. 258

36. Mendenhall, W., and Beaver, R., Introduction to Probability and Statistics, PWS-Kent Publishing Co., Boston, 1991

37. Simon, H., A., *The Sciences of the Artifical*, MIT Press, Cambridge, Massachusetts, 1981

38. Schwefel, H. P., and Manner, R., eds., *Parallel Problem Solving from Nature (PPSN 1)*, Springer-Verlag, 1991

39. Winston, P., *Artificial Intelligence*, 2nd edition, Addison-Wesley, 1984

40. Forrest, S., and Mitchell, M., "Relative Building Block Fitness and the Building Block Hypothesis", in *Foundations of Genetic Algorithms 2*, D. Whitley, ed., Morgan Kaufmann, 1992

41. Forrest, S., and Mitchell, M., "What Makes a Problem Hard for a Genetic Algorithm?" Some Anamolous Results and Their Explanation" to appear in *Machine Learning*

42. Rumelhart, D., and Mclelland, J., *Parallel Distributed Processing*, vol. 1 and vol. 2, MIT Press, 1988

43. Kundu, S., "Lecture Notes in Artificial Intelligence: #3 Learning in Neural Nets", 1988

44. Denning, P. "Genetic Algorithms", in *American Scientist*, vol. 80, Jan.-Feb. 1992, pp. 12-14

45. Klahr, D., et al, ed., *Production System Models of Learning and Development*, MIT Press, 1987

46. Reingold, E., et al, Combinatorial Algorithms: Theory and Practice, Prentice Hall, Englewood Cliffs, New Jersey, 1977

47. Garey, M., and Johnson, D., *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Co., New York, 1979, p. 37

48. ibid., p. 18

49. Bain, L., and Englehardt, M., *Introduction to Probability and Mathematical Statistics*, PW-Kent Publishing, 1987

Appendix A. GA's, BEV, Overpopulation and 3SAT

GA's, BEV, Overpopulation and 3SAT

Thomas Bitterman

Louisiana State University

Baton Rouge, LA 70803

thomas@bit.csc.lsu.edu

(presented at *Evolution as a Computational Process*, Monterey, CA 1992)

Abstract A new learning method for genetic algorithms is introduced that leads to improved performance in solving 3SAT, an NP-complete problem. This method introduces a way to estimate the fitness of a gene before it is evaluated by the objective function, BEV, and a way to cheaply expand the number of genes examined, Overpopulation.

1. 3SAT

Given a set V of boolean variables, |V| = n , we define 3SAT as a boolean expression formed from the conjunction of disjunctions, each disjunction having exactly 3 distinct members from V. For example: V = { a, b, c, d }, f = (a + b + -c)(-b + -c + d) (-a + b + d) and a satisfying truth assignment would be a=T, b=F, c=T, d=T (equivalently, a=1, b=0, c=1, d=1, or 1011) Each clause must have at least one variable evaluate to true for the clause to evaluate to true. This is a subset of the more general satisfiability problem, SAT, but is computationally just as difficult (i.e. both are NP-complete [1]). 3SAT is in a special form called conjunctive normal form, and is also known as 3CNF. Work on k-DNF, another special form where each clause contains k variables and-ed together, with clauses joined by or-s, has been done by ([2][3][4]).

The most challenging problems facing computer scientists today come under the technical name of NP-complete. NP stands for Nondeterministic Polynomial time. In terms of Turing Machines. problems in NP are those that are solvable by a nondeterministic TM in polynomial time. A problem is likely to be in NP if it is hard to discover the solution, but easy to check the validity of any proposed solution. Complete refers to a special quality some problems in NP have, that of being the "hardest" problems in NP. Many NP-complete problems are known, and they are useful in proofs establishing other problems to be in NP. All of the NP-complete problems can be transformed into any other NP-complete problem in polynomial time. Therefore, if any of the NP-complete problems have a polynomial-time solution, they all do. Included in the group of NP-complete problems are the Traveling Salesman Problem, most optimization problems, scheduling, cryptography, and Satisfiability. This makes results in the solution of boolean expressions applicable to a wide range of problems. In addition, boolean representations are a natural way of taking advantage of the efficiency inherent in binary genetic representations.

2. Bitwise Expected Value

Bitwise expected value (bev) is a method whereby the fitness value of each gene that results from the mating process can be estimated before the new gene is evaluated by the objective function. The process is simple: divide the fitness value of each parent gene by its length to find its per-allele fitness. Then, multiply each parent's per-allele fitness by the number of alleles it contributes to the child and sum the products. This is the child's bev. An example of the calculation:

00011*11 fitness value: 140 gene 1

10101*01 fitness value: 70 gene 2

after crossover at *

0001101 bev = (140/7)*5 + (70/7)*2 = 100 + 20 = 120 gene 1'

1010111 bev = (140/7)*2 + (70/7)*5 = 40 + 50 = 90 gene 2'

This assumes that each allele is equally responsible for the fitness of the gene as a whole. This is not always a realistic assumption, but there is no reason that the alleles could not be weighted to emphasize certain alleles known to be important. This would allow natural incorporation of problem-specific knowledge into the genetic search. In addition, the children always have bevs in between the fitness values of their parents, emphasizing the tentative nature of the bev.

This procedure has great versatility. It can work with any form of crossover because all that is needed is the parents' fitness values and how many alleles were taken from

each. Unequal parental gene lengths are likewise easily accommodated. Adaptation to mutation would be accomplished by taking the gene's fitness, dividing it by the total number of alleles in the gene, and then multiplying the result by the number of unmutated alleles. This would assume that mutation is always harmful, and in linear proportion to the number of alleles mutated in a single gene. Bev is a local operator (it needs no global information about strings other than the parents, so it won't hinder parallelization). It can also be adapted to classifier systems, a form of genetic algorithm that functions as an expert system, as Bitwise Expected Productivity. Parallel results will be given later.

3. Overpopulation

Why would one want to know the bev of a gene if it is going to be evaluated by the objective function anyway?

The answer lies in overpopulation. The more genes in the population, the more times the objective function has to be evaluated. It would be great if only the good genes were evaluated. But if we knew what the good genes were before we evaluated them, we would just create only good genes in the first place! Also, if we have some bad luck, it may happen that the good genes we have lose out on the roulette wheel and don't get to propagate good schemas. It would be great if we could select a few more genes to reproduce and thus heighten the odds that the good genes would get chosen. But that would lead to increasing population size and more evaluations of the objective function, leading to exhaustive search.

One way to avoid the loss of good genes is to rank the genes in order of fitness and allocate mating according to rank. This is an efficient way to ensure that good genes don't either lose out or dominate the gene pool early, but it separates the fitness of the gene from it's success at reproduction. Several genes that have very similar, or identical, fitness values can (and probably will) be allocated different amounts of children. This is due entirely to the arbitrary nature of the ranking system.

Another option is to maintain the few best individuals in the population from generation to generation. This way, good genes never die. Unfortunately, this can lead to premature convergence, as the successful gene(s) tend to make the population more homogenous as their progeny come to take over.

A third method would be to just let the population grow with each generation. All genes produced would be allowed to enter the mating pool at any time after they were produced. Nothing is lost, so backtracking is minimized. The evaluation is no more costly, because there are still only n genes produced at each generation, and many of these are bound to be repeats of previously evaluated genes. This method does lead to premature convergence, as the first genes that are successful will tend to be in on most of the mating and smother slower-starting genes. It would also use a lot of memory.

My solution is Overpopulation. In Overpopulation, a certain number M (for Malthus), $M > 1$, is chosen and Mn genes are chosen with replacement from the population (which has size n). These genes undergo crossover in the usual way, with the addition that their bevs are calculated as above. A temporary population is of size Mn is thus created. This temporary population is sorted according to the bev values of the individuals. Only the n individuals with the highest bevs become part of the next generation, ensuring a constant population size.

4. The Application of BEV and Overpopulation to 3SAT

I wrote up a genetic algorithm in C with one point crossover and mutation, and added BEV and Overpopulation to try and improve performance. Crossover rate was set to 0.6, and the mutation rate was 0.01. It appears that BEV and Overpopulation can allow for higher mutation rates than simple classifier systems. After initial success with maximizing some simple functions, both unimodal and ones with local maxima, I looked around for a challenging problem that lent itself to a binary encoding. Translating a problem into bit strings that can productively be manipulated by a genetic algorithm is part of the art of genetic algorithm design. I didn't want the performance of my

new operators to be clouded by coding issues, so a simple coding scheme seemed best. Satisfiability and boolean concept learning had already been tried ([5][6][7][8]) but not 3SAT. 3SAT has lent itself well to my research because of:

- The natural mapping T=1, F=0.
- Uniform clause size made number of clauses satisfied a good fitness rating.
- It was easy to produce test cases.
- The test cases were in NP ([9][10][11]).
- Success was easy to certify, as opposed to a random TSP.
- There were no difficulties with illegal genes, as opposed to illegal tours in a TSP.

5. Results

Solution times for all size expressions tested were very good. (See Fig. 1) The smallest expression tested, which consisted of 30 variables, was solved in an average of less than 23 generations of 60 individuals. An equivalent algorithm which did not use BEV and Overpopulation often failed to find a solution after a 200 generation run. The percentage of the search space actually searched was very small. With 100 variables, the search space size is $1.23*10^{30}$, but the new operators allowed the algorithm to search an average of only 17,126 strings, an extremely small percentage. The percentage would be even lower if duplicate strings generated were not included in the count..

Parallelization of BEV and Overpopulation was surprisingly easy. A SIMD architecture (MasPar DPU Model MP-1208) was chosen. Porting the algorithm, in C, from a VAX 11/780 running UNIX to MPL on a MasPar running Ultrix required the addition of no more than 20 lines of code. The portions benefiting most from the parallel implementation were the objective and fitness functions. Since each gene was compared to the same expression, this expression could be copied onto many different processors, and each gene assigned its own, independent process. In this manner all the genes could be evaluated at once, with the only serial portion being the communication

of each to gene from the main processor to its subordinates. The calculation of bev for each gene took place serially to save communication time, but could also have been done in parallel. The parallel application ran about 10 times faster than the serial program, measured by wall-clock time because both the VAX and the MasPar are time-sharing systems, and it was unclear what would constitute a fair performance comparison between the two.

6. References

1. Garey. M., and Johnson, D., Computers and Intractability, W.H. Freeman and Company, 1979

2. Valiant, L. "A Theory of the Learnable", in Communications of the ACM, vol. 27, no. 11, Nov. 1984, pp. 1134-1142

3. Angluin, D., "Learning Regular Sets from Queries and Counterexamples", in Information and Computation 75, 1987, pp 87-106

4. Kaelbling, L., "Learning Functions in k-DNF from Reinforcement", in Proceedings of the Seventh International Conference on Machine Learning, 1990, pp. 162-169

5. DeJong, K. A., and Spears, "Using Genetic Algorithms to Solve NP-Complete Problems", in Genetic Algorithms: Proceedings of the Third International Conference, 1989

6. McCallum, R., and Spackman, K., "Using Geneic Algorithms to Learn Disjunctive Rules from Examples", in Proceedings of the Seventh International Conference on Machine Learning, 1990

7. Bonelli, P., et al, "NEWBOOLE: A Fast GBML System", in Proceedings of the Seventh International Conference on Machine Learning, 1990

8. Gu, J., "Efficient Local Search for Very Large-Scale Satisfiability Problems", in SIGART Bulletin, vol. 3, no. 1, pp. 8-12

9. Goldberg, A., et al, "Average Time Analysis of Simplified Davis-Putnam Procedures", in Information Processing Letters, vol. 15, no, 2, 1982, pp. 72-75

10. Chvatal, V., and Szemeredi, E., "Many Hard Examples for Resolution", in Journal of the ACM, vol. 35, no. 4, Oct. 1988, pp. 759-768

11. Foulds, L. R., Combinatorial Optimization for Undergraduates, Springer-Verlag, 1984, pp. 170-177

12. Cooper, L., and Steinberg, D., Introduction to Methods of Optimization, W. B. Saunders Co., 1970, pp. 325-327

13. Lawler, E. L., et al, The Traveling Salesman Problem, John Wiley and Sons, 1985

14. Liepins, G., and Wang, L., "Classifier System Learning of Boolean Concepts", in Genetic Algorithms: Proceedings of the Fourth International Conference, 1990, pp. 318-323

Appendix B. Algorithm Descriptions

Name: crossover(chromosome parent1 parent2 child1 child2, int lchrom ncross nmutation jcross, float pcross pmutation xf1 xf2 fit1 fit2)

Input: parent1, parent2, lchrom, ncross, nmutation, jcross, pcross, pmutation, fit1, fit2

Output: child1, child2, ncross, xf1, xf2

Method: if flip(pcross) then

pick crossover spot in first lchrom-1 spots

ncross <- ncross + 1

end then

end if

for j in [0...crossover spot]

put parent1[j] through mutation and into child1[j]

put parent2[j] through mutation and into child2[j]

end for

for j in (crossover spot...lchrom]

put parent1[j] through mutation and into child2[j]

put parent2[j] through mutation and into child1[j]

end for

the expected fitness for the children (xf1 and xf2) is the weighted average of the fitness of the parents

Name: decode(chromosome chrom, int lbits)

Input: chrom, lbits, MAXCLAUSES, num_clauses, expression

Output: the decoded value of chrom, which is the number of clauses it satisfies

Method: for i in [0...MAXCLAUSES)

```
    true_clauses[i] <- 0
  end for
  for i in [0...lbits)
    for j in [0...num_clauses)
      for k in [0...3)
        if (chrom[i] and (expression[j][k] = ( i +1 )) then true_clauses[j] <- 1
        if (!chrom[i] and (expression[j][k]=( -i - 1)) then true_clauses[j] <- 1
      end for
    end for
  end for
  for j in [0...num_clauses)
    sum <- sum + true_clauses[j]
  end for
  return( sum )
```

Name: init_clauses

Input: none

Output: target boolean vector and a boolean expression with num_clauses conjunctive clauses, each of which contains three variables, which is satisfied by the target vector.

Method: for i in [1...lchrom]

```
    true_chrom ← flip(0.5)
  end for
  for i in [1...num_clauses]
    if true_chrom[i] then expression[i-1][0] ← i
                     else expression[i-1][0] ← -i
  j = rnd( 1, lchrom )
  if flip( 0.5 ) then expression[i-1][1] ← j
                 else expression[i-1][1] ← -j
  if flip ( 0.5 ) then expression[i-1][2] ← j
                  else expression[i-1][2] ← -j
```

Name: select(int popsize, float sumfitness, population pop)

Input: popsize, sumfitness, pop

Output: the index of a gene, the probability of a gene's index being picked increases directly with its fitness

Method: rg ← randomg * sumfitness

```
j ← 0
while ( ( partsum < rg ) and ( j < popsize ) ) do
  partsum ← partsum + pop[j].fitness
  j ← j + 1
end while
return( j )
```

Appendix C. Code

```
/* takes the top n indivividuals from the temporary population and transfers them to the
new population */
void g2()
{
        int j;

        sort();
        for ( j = 0; j , popsize; j++ )
        {
                chrom_eq( &newpop[j], tp[j] );
                newpop[j].x = tp[j].x;
                newpop[j].fitness = tp[j].fitness;
                newpop[j].parent1 = tp[j].parent1;
                newpop[j].parent2 = tp[j].parent2;
                newpop[j].xsite = tp[j].xsite;
                newpop[j].exp_fitness = 0;
        }
}

/* selects the genes that will undergo crossover and mutation and calls the requisite
function "crossover". on return it performs some bookeeping */
void generation()
{
        int j, mate1, mate2, jcross;
        float exp_fit1, exp_fit2;

        for ( j = 0; j < 2*popsize; j+=2 )
        {
                mate1 = select(popsize, sumfitness, oldpop);
                mate2 = select(popsize, sumfitness, oldpop);
                crossover( oldpop[mate1].chrom, oldpop[mate2].chrom, tp[j].chrom,
                                tp[j+1].chrom, lchrom, &ncross, &nmutation, &jcross,
                                pcross, pmutation, &exp_fit1, &exp_fit2,
                                oldpop[mate1].fitness, oldpop[mate2].fitness );
                tp[j].x = decode(tp[j].chrom, lchrom);
                tp[j].fitness = objfunc(tp[j].x);
                tp[j].parent1 = mate1;
                tp[j].parent2 = mate2;
                tp[j].xsite = jcross;
                tp[j].exp_fitness = exp_fit1;
                tp[j+1].x = decode(tp[j+1].chrom, lchrom);
                tp[j+1].fitness = objfunc(tp[j+1].x);
                tp[j+1].parent1 = mate1;
                tp[j+1].parent2 = mate2;
                tp[j+1].xsite = jcross;
                tp[j+1].exp_fitness = exp_fit2;
                }
        g2();
}
```

```
/* performs crossover on the genes parent1 and parent2 and calculates bev for the
children */
void crossover( parent1, parent2, child1, child2, lchrom, ncross, nmutation, jcross,
                        pcross, pmutation, xf1, xf2, fit1, fit2 )
chrom parent1, parent2, child1, child2;
int lchrom, ncross, nmutation, nmutation, jcross;
float pcross, pmutation, xf1, xf2, fit1, fit2;

{
        int j, spot;

        if ( flip( pcross ) )
        {
                *jcross = rnd(0, lchrom-2);
                (*ncross)++;
        }
        else *jcross = lchrom - 1;
        spot = *jcross + 1;
        for ( j = 0; j <= *jcross; j++ )
        {
                child1[j] = mutation(parent1[j], pmutation, &nmutation);
                child2[j] = mutation(parent2[j], pmutation, &nmutation);
        }
        if (*jcross != lchrom - 1)
        {
                for ( j = (*jcross) + 1; j < lchrom; j++ )
                {
                        child1[j] = mutation(parent1[j], pmutation, &nmutation);
                        child2[j] = mutation(parent2[j], pmutation, &nmutation);
                }

        }
        *xf1 = (spot * fit1 + (lchrom - spot) * fit2) / lchrom;
        *xf2 = (spot * fit2 + (lchrom - spot) * fit1) / lchrom;
}

/* returns the value of the input allele after applying mutation with probability
pmutation */
allele mutation( alleleval, pmutation, nmutation )
alele allelval;
float pmutation;
int *nmutation;

{
        mutate = flip(pmutation);
        if (mutate)
        {
                (*nmutation)++;
                return( !alleleval );
        }
        return( alleleval );
}
```

```
/* implements the roullette wheel method of selection */
int select( popsize, sumfitness, pop )
int popsize;
float sumfitness;
population pop;

{
        float rg, partsum = 0;
        int j = 0;

        rg = randomg() * sumfitness;
        for ( j = 0; ((partsum<rg) && (j<popsize)); j++ )
        {
                partsum += pop[j].fitness;
        }
        return(j-1);
}

/* a sample set of initial parameters. could be typed in and compiled, or read from an
input file */
void initdata()
{
        char ch;
        int j;

        popsize = 40;
        lchrom = 20;
        maxgen = 200;
        pcross = 0.6;
        pmutation = 0.01;
        num_clauses = 1000;
        nmutation = 0;
        ncross = 0;
}

/* initializes the population to a random set of starting genes */
void initpop()
{
        int j, j1;

        for ( j = 0; j < popsize; j++ )
        {
                for ( j1 = 0; j1 < lchrom; j1++ )
                {
                        oldpop[j].chrom[j1] = flip(0.5);
                }
                oldpop[j].x = decode(oldpop[j].chrom, lchrom);
                oldpop[j].fitness=objfunc(oldpop[j].x)
                oldpop[j].parent1 = oldpop[j].parent2=oldpop[j].xsite=0;
        }
}
```

```
/* returns the objective function value corresponding to the fitness value of a gene. not
used, but here for expandability */
float objfunc( x )
int x;

{
        return((float)x);
}

/* the fitness function. returns the number of clauses the gene satisfies */
int decode( chrom, lbits )
chromosome chrom;
int lbits;

{
        int i, j,k;
        int true_clauses[num_clauses];

        for ( i = 0; i < num_clauses; i++ )
        {
                true_clauses[i] = 0;
        }
        for ( i = 0; i < lbits; i++ )
                for ( j = 0; j < num_clauses; j++ )
                        for ( k = 0; k < 3; k++ )
                        {
                        if ((chrom[i])&&(expression[j][k]==(i+1))) true_clauses[j]=1;
                        if ((!chrom[i])&&(expression[j][k]==(-i-1))) true_clauses[j]=1;
                        }
        for ( j = 0; j < num_clauses; j++ )
                sum += true_clauses[j];
        return(sum);
}

/* randomly initializes the target truth assignment and the target clauses */
void init_clauses()
{
        int i, j;

        for ( i = 0; i < lchrom; i++ )
                true_chrom[i] = flip(0.5);
        for ( i = 1; i < num_clauses+1; i++ )
        {
                if (true_chrom[i]) expression[i-1][0] = i;
                else expression[i-1][0] = -i;

                j = rnd( 1, lchrom );
                if (flip(0.5)) expression[i-1][1] = j
                else expression[i-1][1] = -j;
                j = rnd( 1, lchrom );
                if (flip(0.5)) expression[i-1][2] = j
                else expression[i-1][2] = -j;
        }
}
```

```
/* prints out a chromosome */
void writechrom( chrom, lchrom )
chromosome chrom;
int lchrom;

{
        int i;

        for ( j = 0; j < lchrom; j++ )
        {
                if (chrom[j]) printf("1");
                else printf("0");
        }
        printf("\n");
}

/* reports the first generation a satisfying gene is found in. also end the program at atht
time or if the preset generation limit is reached */
void report( gen )
int gen;

{
        int j;

        for ( j = 0; j < popsize; j++ )
        if ((oldpop[j].fitness == num_clauses) || (gen == 200))
        {
                printf("%d\n", gen);
                exit(0);
        }
}

/* class all the initialization functions */
void initialize();
{
        initpop();
        initclauses();
        statistics( popsize,  &sumfitness, oldpop );
}

/* formerly a multi-purpose statistic-gathering function, it now only computes the sum
of fitness values for the whole population */
void statistics( popsize, sum, oldp )
int popsize;
float *sum;
population oldp;
{
        int i = 0;
        float temp = 0;

        for ( i = 0; i < popsize; i++ )
                temp += oldp[i].fitness;
        *sum = temp;
}
```

```
/* sets the contents of chromosome a to the contents of chromosome b */
void chrom_eq( a, b )
struct individual *a, b;

{
        int i;
        struct individual temp;

        for ( i = 0; i < lchrom; i++ )
                temp.chrom[j] = b.chrom[j];
        *a = temp;
}

/* switches the places of two entire individuals in the population */
void switcher( a, b )
int a, b;

{
        struct individual temp, temp1, temp2;

        temp1 = tp[a];
        temp2 = tp[b];
        chrom_eq(&temp, temp2);
        temp.x = temp2.x;
        temp.fitness = temp2.fitness;
        temp.parent1 = temp2.parent1;
        temp.parent2 = temp2.parent2;
        temp.xsite = temp2.xsite;
        temp.exp_fitness = temp2.exp_fitness;

        chrom_eq(&temp2, temp1);
        temp2.x = temp1.x;
        temp2.fitness = temp1.fitness;
        temp2.parent1 = temp1.parent1;
        temp2.parent2 = temp1.parent2;
        temp2.xsite = temp1.xsite;
        temp2.exp_fitness = temp1.exp_fitness;

        chrom_eq(&temp1, temp);
        temp1.x = temp.x;
        temp1.fitness = temp.fitness;
        temp1.parent1 = temp.parent1;
        temp1.parent2 = temp.parent2;
        temp1.xsite = temp.xsite;
        temp1.exp_fitness = temp.exp_fitness;

        tp[a] = temp1;
        tp[b] = temp2;
}
```

```
/* sorts the temporary population by bev */
void sort()
{
        int i, j;

        for ( i = 0; i < 2*popsize; i++ )
                for ( j = i+1; j < 2*popsize; j++ )
                        if ( to[i].exp_fitness < tp[j].exp_fitness ) switcher(i, j);
}

/* the main loop */
main()
{
        int seed;

        oldp = (population *)op;
        newp = (population *)np;
        fp = fopen("infile", "r");
        fscanf(fp, "%d", &seed);
        fclose(fp);
        system("rm infile");
        fp = fopen("infile", "w");
        fprintf(fp, "%d\n", seed+1);
        fclose(fp);
        srandom(seed);
        gen = 0;
        initialize();
        for ( gen = 1; ; gen++ )
        {
        generation();
        statistics( popsize, &sumfitness, oldpop );
        report(gen);
        tempp = oldp;
        oldp = newp;
        newp = tempp;
        }
}
```

```
/* some sample definitions and variable assignments */
sga_defs.h
#define MAXLENGTH 100
#define MAXPOP 400

typedef int allele;
typedef allele chromosome[MAXLENGTH];
struct individual
{
        chromosome chrom;
        int x;
        float fitness;
        int parent1, parent2;
        int xsite;
        float exp_fitness;
}
typedef individual population[MAXPOP];
population op, np, oldpop, newpop, tp;
population *newp, *oldp, *tempp;
float sumfitness;

#define num_clauses 1000
#define num_vars 20
#define MAXCLAUSES 5000
```

Vita

Thomas Bitterman was born in 1965 in Cleveland, Ohio. He graduated from Padua Franciscan High School in Parma, Ohio in 1984. He attended Kent State University, receiving his B.S. in Computer Science in 1988. He then attended Louisiana State University from 1988-1993, receiving his Ph.D. in Computer Science. His main research interests are genetic algorithms, artificial intelligence, and theory of computation. He also holds a black belt in ju-jutsu.

DOCTORAL EXAMINATION AND DISSERTATION REPORT

Candidate: Thomas Bitterman

Major Field: Computer Science

Title of Dissertation: Genetic Algorithms and the Satisfiability of Large Scale Boolean Expressions

Approved:

Major Professor and Chairman

Dean of the Graduate School

EXAMINING COMMITTEE:

Date of Examination:

2/18/93