01 May 1993

Considerations for Rapidly Converging Genetic Algorithms Designed for Application to Problems with Expensive Evaluation Functions

Richard Patrick Rankin

Ralph W. Wilkerson
Missouri University of Science and Technology, ralphw@mst.edu

CONSIDERATIONS FOR RAPIDLY CONVERGING GENETIC ALGORITHMS DESIGNED FOR APPLICATION TO PROBLEMS WITH EXPENSIVE EVALUATION FUNCTIONS

R. Rankin* and R. W. Wilkerson

CSc-93-12

Department of Computer Science

University of Missouri - Rolla

Rolla, MO 65401 (314)341-4491

*This report is substantially the Ph.D. dissertation of the first author, completed May 1993.

ABSTRACT

A genetic algorithm is a technique designed to search large problem spaces using the Darwinian concepts of evolution. Solution representations are treated as living organisms. The procedure attempts to evolve increasingly superior solutions. As in natural genetics, however, there is no guarantee that the optimum organism will be produced.

One of the problems in producing optimal organisms in a genetic algorithm is the difficulty of premature convergence. Premature convergence occurs when the organisms converge in similarity to a pattern which is sub-optimal, but insufficient genetic material is present to continue the search beyond this sub-optimal level, called a local maximum.

The prevention of premature convergence of the organisms is crucial to the success of most genetic algorithms. In order to prevent such convergence, numerous operators have been developed and refined. All such operators, however, rely on the property of the underlying problem that the evaluation of individuals is a computationally inexpensive process.

In this paper, the design of genetic algorithms which intentionally converge rapidly is addressed. The design considerations are outlined, and the concept is applied to an NP-Complete problem, known as a Crozzle, which does not have an inexpensive evaluation function. This property would normally make the Crozzle unsuitable for processing by a genetic algorithm. It is shown that a rapidly converging genetic algorithm can successfully reduce the effective complexity of the problem.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

I. INTRODUCTION

A. GENERAL INTRODUCTION

Genetic algorithms are an interesting class of problem solving techniques loosely based on Darwinian concepts of evolution. The literature indicates that these algorithms have been applied to a large number of problems, especially NP-Complete problems with varying degrees of success [Goldberg (1989c)].

Genetic algorithms (GAs) are able to randomly sample large areas of a problem search space. They then evolve new search points based upon the performance of the old search points in hopes of improving the performance of the overall search.

The abstract characteristics of GA implementations are very consistent throughout the reports in the literature although specific details of implementations vary widely. These characteristics include operators designed to prevent the GA from converging to solutions too rapidly. This premature convergence is discussed at great length in the literature. The prevention of this undesired convergence is aimed at forcing the genetic algorithm to continue searching the problem search space without falsely being trapped at local maxima. This continued searching, however, assumes that the ability to evaluate the performance of the GA is an inexpensive proposition.

This project explores the design considerations in creating a genetic algorithm which does, intentionally, converge rapidly. To illustrate the performance of these ideas, an NP-Complete crossword puzzle game, the Crozzle, is used as an illustration. The evaluation of search points in the Crozzle search space do not have an inexpensive evaluation function. Therefore, if a genetic algorithm is to be used, it cannot be utilized in the "normal" way. It is too expensive to continually evaluate search points and, therefore, the algorithm must converge rapidly.

B. THE CROZZLE

The Crozzle is a word game based upon the construction of crossword puzzles. The most familiar type of crossword puzzle is the constrained crossword puzzle. These puzzles appear in many magazines and newspapers, and consist of a grid with black squares, empty squares and clues. One uses the clues to determine the words which fit into the crossword puzzle, inserting letters only in the empty squares.

Constrained crossword puzzles are considered constrained because of the presence of black squares in the grid when one begins working on the puzzle. The presence or absence of clues does not affect whether or not a puzzle is constrained.

Unconstrained puzzles are crossword puzzles where there are no black squares in the grid when one begins to solve the puzzle. An unconstrained puzzle can be either completely interlocked (no black squares are allowed in the solution), or the rules may allow the insertion of black squares by the puzzle solver during the solution phase. A completely interlocked puzzle is shown in Figure 1. The lexicon consisted of the words {abbas, araca, racon, ovoid, nanny, aaron, brava, bacon, acoin, sandy}.

A solution to a crossword puzzle is a grid which has been completely filled in according to the general rules of crossword puzzle construction. In the familiar

A	A	R	O	N
B	R	A	V	A
B	A	C	O	N
A	C	O	I	N
S	A	N	D	Y

Figure 1. Completely Interlocked Crossword

constrained puzzles, only a single solution might exist using the clues. If the clues are disregarded, however, there typically are large numbers of solutions which exist based upon the remaining rules. An example of a constrained puzzle with multiple solutions is shown in Figure 2 [Ginsberg (1990)]. This grid has yielded over 10,000,000 solutions given a specific lexicon containing only approximately 1500 words. A procedure for estimating the number of solutions for a given puzzle has been published [Harris (1992d)].

In an unconstrained crossword puzzle, there are large numbers of grid configurations which must be explored, as well as potentially large numbers of solutions to each of those grids. An unconstrained grid with one hundred squares, for example, has 2^{100} grid configurations. Each square has the possibility of being a black square or an empty, usable square.

There have been few published accounts regarding the mechanical solution of crossword puzzles. Although Mazlack is generally credited with the first attempts at automated solutions to crossword puzzles, his reported efforts are considered

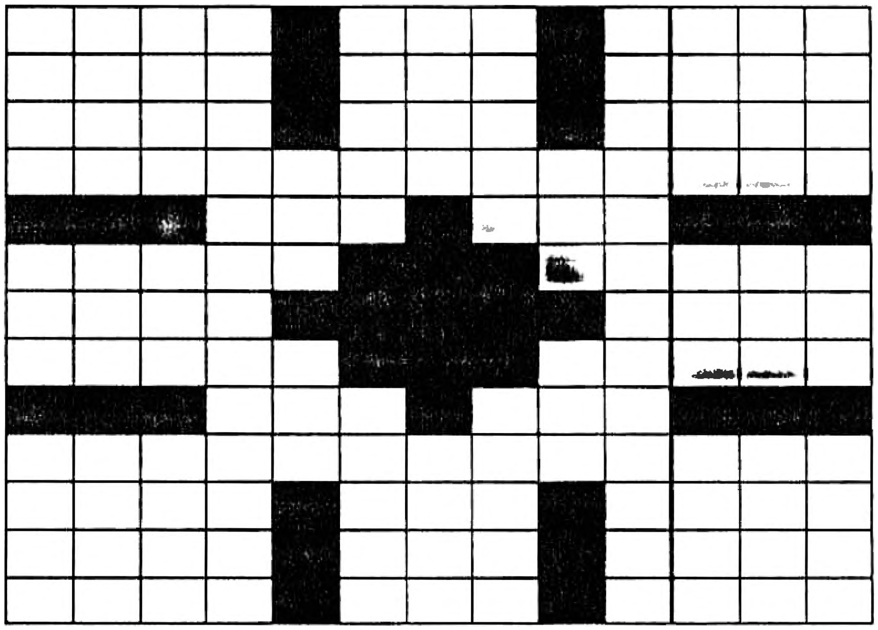

Figure 2. Constrained Crossword Puzzle

unimpressive [Mazlack (1976)]. The first successful attempt at crossword puzzle construction is credited to Smith and Steen [Smith (1981)]. Their published attempts at crossword puzzle solution, however, generally concern only constrained crossword puzzles.

The Crozzle is an unconstrained crossword puzzle with a required domain of words used in the solution and a unique scoring system. The puzzle is published regularly, in recent years, monthly, in *The Australian Women's Weekly* magazine. The Crozzle is published as a contest for the readers. The goal is to take the grid and a word list, build a solution according to the construction rules, and maximize the score based upon the scoring rules in effect.

To date, no computer program has been able to win the Crozzle contest. Several published accounts exist discussing various automated attempts to win Crozzle contests [Harris (1990b), Harris (1992a), Harris (1992c), Harris (1993b), Rankin (1993b), Rankin (1993a)]. Due to the large search space inherent in the problem of Crozzle solution and a time limit on the contest, automated efforts aimed at winning the Crozzle have failed. The goal of this project was to apply a new rapidly converging genetic algorithm to the Crozzle in an attempt to increase the chances of an automated program successfully winning the contest.

C. GENETIC ALGORITHMS

Genetic algorithms are a type of heuristic search technique (Goldberg [1989c]). These algorithms, while not a random search, strongly rely on random numbers. If one views the solution space to a problem as a three dimensional space with maximum values represented as peaks rising from a plane, then the idea behind the genetic algorithm is to sample a large number of data points on the surface. Each point is rated in terms of its value, to determine if that particular point might be near a maximum value. The points with higher scores are selected for more experimentation. Points with lower scores are discarded. Therefore, the algorithm attempts to do hill-climbing in the search space by first attempting to find large numbers of hills to check, then gradually focussing on the better locations.

There are five components, discussed in detail below, required at the abstract level for a genetic algorithm:

1. a chromosomal representation of solutions to the problem,
2. a way to create an initial population of solutions,
3. an evaluation function that plays the role of the environment, rating the solutions in terms of their "fitness",
4. genetic operators that alter the composition of the children during reproduction, and
5. values for the parameters that the genetic algorithm uses (population size, probabilities of applying genetic operators, etc) [Davis (1987)].

A genetic algorithm is different from more commonly known search techniques. The original concept behind a genetic algorithm is that it should be independent of the domain of the specific problem. This frequently does not bear out in practice. There are some characteristics, however, that are in common to all genetic algorithms. These characteristics relate to the coding of the parameters, the number of points examined, and the transition rules.

A genetic algorithm works with the encoding of parameters and not the parameters themselves. It does not know what the encoding represents. The meaning of the encoding is not necessary for the operation of the GA.

An entire population of points within the search space is used. One does not choose a single point and attempt to optimize from that point. One chooses large numbers of points and explores those which seem to offer the most promise of a "good" solution.

The transition rules are probabilistic and not deterministic. This does not imply that they are random walks through the search space, however. Points are, initially, randomly selected for examination. The GA operators are also based on random numbers, but they are applied to chromosomes which already represent points which appear of interest.

D. THE PROBLEM

At first glance, it is easy to underestimate the difficulty of generating the winning solution to a Crozzle puzzle. The grid contains only 150 empty squares and the allowable word list is only slightly over one hundred words generally. The search space, however, is extremely large - much too large to do a complete traversal. The search space is a function of both the number of words in the word list and the size of the empty grid.

Attempts have been made to estimate the number of nodes in the search tree for a typical Crozzle. For example, a Crozzle was randomly selected. During the traversal of the search space, the number of nodes at various levels in the search tree were counted [Harris (1992a)]. Considering that the typical solution at that time might have thirty words in the solution and the experimental observation that the average fan-out of a node at the higher levels of the tree was approximately ten, an upper bound of 10^{30} nodes can be reasonably accepted. It should be noted, however, that at deeper levels of the search tree, the fan-out can be considerably less than ten. In that paper, however, the authors also present two alternate methods of calculating an estimate of the number of nodes in the same Crozzle. One of these methods yields a lower bound of 10^{19} nodes. The other yields an estimate of 10^{24} nodes. Current Crozzle implementations process around 2000 nodes per second. This means, that to totally traverse the search space of a Crozzle would require approximately sixteen million years.

From experimental results, it appears that almost any word list of fifteen words will fit into the normal Crozzle grid. With a normal list of 110 words, there are ${}^{110}C_{15}$ combinations of words, including a large number of repetitions, that would generate roughly 10^{30} nodes for the Crozzle. This same argument, however, does not apply to solutions with, say twenty-five, words in them. Word lists of that length, again, from experimental observations, will not necessarily all fit into the empty Crozzle grid. At some point, the size of the grid enforces a saturation point.

Table I [Rankin (1993a)] shows the effects of varying both the grid size and the numbers of words available in the lexicon, holding the number of rows constant at ten. It indicates that both the size of the grid and the number of words affect the number of nodes in the search tree. As can be noted from the table, increasing the number of words by five from ten to fifteen, or fifteen to twenty, commonly increases the number of nodes several times. When one increases the number of words from twenty to twenty-five, however, the number of nodes increases only by approximately 1/4. This implies that there is a possible saturation point in the 10 column by 5 row grid somewhere in the range of twenty to twenty-five words. From experimental evidence, when using a full Crozzle grid and a larger lexicon, each five words added to the word list increases the search space by approximately one order of magnitude.

Table I. EFFECT OF GRID AND LEXICON SIZES

Number of Words	LENGTH OF GRID						
	5	6	7	8	9	10	11
10	$3.7*10^3$	$1*10^4$	$3.3*10^4$	$6.9*10^4$	$9.7*10^4$	$1.1*10^5$	$2.0*10^5$
15	$1.7*10^4$	$7*10^4$	$2.7*10^5$	$9*10^5$	$2.0*10^6$	-	-
20	$4.0*10^4$	$5.3*10^5$	$2.6*10^6$	$1.4*10^7$	$5.7*10^7$	-	-
25	$5.5*10^4$	$1.2*10^6$	$6.7*10^6$	-	-	-	-

The highest score possible for any given Crozzle word list is currently unknown. To date, no solution has ever been discovered with a point score higher than the winning solution published by *Australian Woman's Weekly*. The published winning solution shall be referred to as the *Human Winning Solution*, HWS. The word list, configuration, and score for each HWS is considered to be the global maximum score for any Crozzle discussed below. It seems unlikely that humans are routinely discovering the maximum score from a search space of approximately 10^{20} nodes, yet no published solution has ever been exceeded, although several have been tied.

E. THE GOAL OF THE PROJECT

The goal of this project is to eliminate words from the given Crozzle word lists without losing the words required to attain the winning score. The specific words to be eliminated are the words with lengths of six or greater. There was no attempt to trim words of length three, four or five from the word lists. The reasoning behind ignoring the words with lengths of three, four and five is a result of discussing the methods used to solve the Crozzle by human players. These players indicated that the shorter words are not used to develop an overall skeletal structure while constructing solutions. Instead, these words are used opportunistically and players insert them in available positions after a rough solution is completed. The basic frameworks of the human solutions were generated as much as possible from longer words, since these longer words provide more letter positions from which to play additional words. The shorter words were "tucked in" wherever they seemed to fit. These word sets, which appear in every Crozzle puzzle, are called *345 words*.

The "solution" to a Crozzle puzzle requires that a complete grid with interlocked words be generated. The generation of this solution is the portion of the problem which requires enormous amounts of time. For example, even when the exact subset of words is known for a particular solution, it might require twenty or thirty minutes to generate the correct solution. When additional words are added, the time required increases at a rapid rate.

Since there are existing Crozzle solution generators which can generate winning solutions in a reasonable time, given a small enough word list, the Crozzle Solvers are not actually used to test the quality of the solutions created by the GA programs. By using historical data, the trimmed word list produced by this project can be compared to the known solution. If all the words with lengths greater than five are present in the GA-generated sublist, then it is assumed that the existing Crozzle solution generators could find that solution.

II. GENETIC ALGORITHMS

A. BASIC INFORMATION

1. Terms. The terminology used in genetic algorithms is based upon genetics to a large extent, although there are differences in the way some terms are used. A *chromosomal representation* , or a *chromosome*, is basically a string of numbers or bits, depending upon the representation chosen for a particular implementation. Each position in the chromosome is considered a *gene*. The value of a gene is called an *allele*. The position of a gene within a chromosome is its *locus*. An entire collection of chromosomes is called a *genotype*. The value returned by the evaluation function for a chromosome is its *fitness*.

2. Operators. There are three basic operators used in a simple genetic algorithm. These operators have been examined extensively in the literature and numerous variations proposed. The simplest versions are discussed below.

a. Reproduction. Reproduction as implemented in a genetic algorithm is more like a "survival of the fittest" procedure rather than reproduction as normally viewed. The reproduction phase of the algorithm strictly determines which individual chromosomes in the population survive into the next phase.

There are several methods available to implement the reproduction procedure. The details of these will be discussed further below. The abstract view, however, is similar in all of the methods. Each individual chromosome in the population is given a fitness value. The fitness values for the entire population are summed. The ratio of a particular chromosome to the fitness sum of the entire population is its probability of surviving to the next phase. Thus, the probability of chromosome *i* with a fitness value of f_i, and a population value of Σf, is $f_i/\Sigma f$.. For example, if chromosome *i* has a value of 25 and the fitness value of the entire population sums to 100, then the probability of chromosome *i* surviving to the next generation is 25%. In practice, however, what this means is that 25% of the next generation of chromosomes should be copies of chromosome *i*. The problem of achieving this goal is discussed further below.

b. Crossover. The operator, crossover, is more similar to what is often called reproduction than is the reproduction operator. The crossover operator works on the new population generated from the reproduction phase. The members of this new population are "mated" by combining genes from two parents to create two new offspring.

Traditional one-point crossover requires that two chromosomes be selected at random from the population. A random number in the interval [1,len-1], where len is the length of a chromosome, is generated. This is called the *crossover site*. For a crossover site *c*, two new offspring are created by swapping the genes in the parents in the loci r+1 through len. For example, given two randomly selected chromosomes A and B, with len = 6, and *r* = 2, the new offspring A' and B' would be created:

$$A = a_1\ a_2\ a_3\ a_4\ a_5\ a_6$$
$$B = b_1\ b_2\ b_3\ b_4\ b_5\ b_6$$

$$A' = a_1\ a_2\ a_3\ b_4\ b_5\ b_6$$
$$B' = b_1\ b_2\ b_3\ a_4\ a_5\ a_6$$

c. Mutation. A frequent problem in many genetic algorithms is that of premature convergence. The idea behind the algorithm is to randomly attack the search space and then to converge the search towards those points which seem to represent the best solutions. Sometimes, however, this convergence process can cause the algorithm to concentrate on some well-fit local maximum and not find the global maximum. Since the chromosomes incorrectly converge on the local maximum, this phenomenon is called *premature convergence.*

In order to prevent premature convergence to local maxima, or at least to minimize the effect, the mutation operator is used. A mutation operator takes an existing new population and randomly changes some of the bits in the chromosome according to a pre-set probability. For example, suppose there are 100 chromosomes involved in the genotype, with 20 bits encoded per chromosome. If the mutation rate is set at 0.002, then 4 bits (100*20*0.002) would be randomly selected and their values altered.

The idea behind mutation is that random bit changes alter the information reflected in a chromosome, and, since the chromosomes represent points in the search space, change the area of focus in the search space for those chromosomes. If a population has prematurely converged upon a local maximum, then the mutation operator will hopefully throw the mutated point back out into a new region of the search space. If the mutated chromosome finds a new local maximum, or the global maximum, then the population should eventually converge to the new point with the higher fitness level.

3. Operation. The operation of a genetic algorithm is straightforward. It is an initialization routine, followed by an iterative loop applying genetic operators. The sequence of operations is shown below in Table II.

Table II. OPERATIONAL SEQUENCE

1) Create initial population
2) Initialize the population with values from the search space (solutions)
3) Evaluate the population
4) Repeat until completed:
    a) Apply reproduction operator
    b) Apply crossover operator
    c) Apply mutation operator
    4) Evaluate the population

4. Example. Tables III and IV show a simple example of a genetic algorithm in action [Goldberg (1989c)]. The evaluation function, which is to be maximized for x in the closed interval [0, 31] is $f(x) = x^2$. Table III shows the initial, randomly generated chromosomes in the left hand column, represented in binary form. The evaluation function, in this example, merely translates the binary representation of the chromosome (given in the "X Value" column,) then squares that value. The result of the evaluation function for each chromosome is shown in the fourth column. The number shown in that column is the chromosome's fitness.

Most genetic algorithms maintain a stable number of chromosomes in the population. In this example, the population consists of four individuals, and will therefore stay at four individuals in future generations. To create the next generation, the reproduction operator is applied. The function "pselect" shows the ratio of each individual's fitness to the entire populations' fitness. This gives the likelihood that that individual will be duplicated in the reproduction phase. The "Expected Count" column shows how many copies of each individual would be expected to be in the next generation. This count is merely the population size (four in this example) multiplied by the value of pselect for that individual. Obviously, "portions" of individuals cannot survive. Therefore, the actual count indicates the number of copies of each individual actually surviving the reproduction phase. String number 3, which had a very poor fitness evaluation, has been eliminated. String number 2, which had a high fitness rating, received two copies of itself in the next generation.

Table IV shows the situation after the reproduction phase is completed in the left hand column. As can be seen, the original string numbered 3 is not present, and there are two copies of string number 2. At this point, the crossover operator is applied.

The crossover operator selects two chromosomes from the population as operands. The "Mate" column shows those selected as pairs. Next, for each pair, a crossover site is randomly selected. This value is given in the "Site" column. After crossover, the resulting individuals are shown in the "New Population" column. Following the operational flow shown above, the mutation operator would be applied next. A mutation operator is frequently expressed as mutations per thousands of bits, however, and the small number of bits in this example would clearly have a very low likelihood of mutation. Therefore, the mutation operator is not actually applied.

Table IV goes on to show the "X Value" and evaluation function results when used on the new population. As can be seen , the fitness of the entire population (sum), the average fitness of the population, and the level of the individual with the highest level of fitness have all increased after only a single generation. This procedure would be repeatedly applied for either a certain number of iterations, or until a certain value has been attained.

Table III. GA EXAMPLE 1

String #	Initial Population	X value	f(x)=x^2	pselect	Expected Count	Actual Count
1	0 1 1 0 1	13	169	0.14	0.58	1
2	1 1 0 0 0	24	576	0.49	1.97	2
3	0 1 0 0 0	8	64	0.06	0.22	0
4	1 0 0 1 1	19	361	0.31	1.23	1
sum			1170	1.00	4.00	4.00
Average			293	0.25	1.00	1.00
Max			576	0.49	1.97	2.00

Table IV. GA EXAMPLE 2

Pool After Reproduction	Mate	Site	New Population	X Value	f(x) = x^2
0 1 1 0 / 1	2	4	0 1 1 0 0	12	144
1 1 0 0 / 0	1	4	1 1 0 0 1	25	625
1 1 / 0 0 0	4	2	1 1 0 1 1	27	729
1 0 / 0 1 1	3	2	1 0 0 0 0	16	256
Sum					1754
Avg.					439
Max					729

B. LITERATURE REVIEW

1. What is a Genetic Algorithm? A genetic algorithm is one of a family of adaptive search techniques. They are loosely based on the idea of the mechanics of natural selection and genetics. The basic idea is to have a population of individual "creatures" represented in a computer program. These individuals then are subjected to a process which includes the survival of the fittest, reproduction, and mutation.

> GAs derive their name from the fact that they are loosely based on models of genetic change in a population of individuals. These models consist of three basic elements: (1) a Darwinian notion of "fitness" which governs the extent to which an individual can influence future generations; (2) a "mating operator" which produces offspring for the next generation; and (3) "genetic operators" which determine the genetic makeup of offspring from the genetic material of the parents. [De Jong (1988)]

The key element of genetic algorithms (GA's) is that they search large spaces with a wide range of samplings. The samplings indicating better solutions, i.e. fitter individuals, are used to move more samplings to that area of the search space.

Traditional optimization and search techniques can be classified as calculus-based, random, and enumerative [Goldberg (1989c)]. Calculus-based techniques are further divided into direct and indirect techniques.

Direct calculus-based techniques for a given function work by selecting a point in the search space, and following the steepest gradient. This method is also known as *hill-climbing*. *Indirect calculus-based techniques* seek to determine local extrema and commonly work by solving usually non-linear equations resulting from setting the gradient to zero. This zero gradient would represent either a local maximum or minimum.

Enumerative techniques are conceptually simple in that they merely examine every point in the search space one at a time. For a problem of non-trivial size, this method is obviously not a viable alternative. *Random search algorithms* basically examine points in the search space randomly and save information relating to the best solution found so far in the process. Once again, given a large enough search space, this method is unlikely to converge. Where T is the number of trials, and S is the points in the search space, would have only a T/S probability of finding the true maximum

Genetic algorithms rely heavily on random number generation. They are, however, a random search technique and not a random search. Simulated annealing is another popular randomized technique. The difference lies in the fact that genetic algorithms randomly select initial search points, but use the resulting feedback to exploit those points with more perceived potential for being near a maximum.

The required parts of a genetic algorithm are discussed below. A general introduction and a simple example may be found above.

Genetic algorithms are being used in a large variety of problem domains. Goldberg lists several pages of projects which have been attempted with genetic algorithms [Goldberg (1989c)]. Other problems include image interpretation [Hill (1992)], geophysics [Sambridge (1992)], school bus routing [Thangiah (1992)] and various design considerations [Pham (1991), Goldberg (1991), Szarkowicz (1991)].

2. <u>Population Sizes</u>. One obvious consideration when developing a genetic algorithm is the size of the population involved. There must be sufficient individuals to randomly sample the search space, but not so many individuals that the population size approaches the number of nodes in the search tree.

> Choosing the population size for a genetic algorithm (GA) is a fundamental decision faced by all GA users. On the one hand, if too small a population size is selected, the genetic algorithm will converge too quickly, with insufficient processing of too few schemata. On the other hand, a population with too many members results in long waiting times for significant improvement, specially when the evaluation of the individuals within a population must be performed wholly or partially in serial: the population is too large to get enough mixing of the building blocks per unit of computation time [Goldberg (1989b)].

Schaffer outlines the then current "state of the art" in population sizing and reaches different conclusions from those of Goldberg. According to De Jong, in 1975, the optimal population size was 50 - 100 individuals. Grefenstette, in 1986, proposed a population size of 30. Goldberg, in 1985, proposed an approximate ideal population size of $pop = 1.65 * 2^{0.21 * length}$ where length is the number of binary digits required for each individual. Using Goldberg's suggestion, if one assumed that the chromosomes in the current project were binary encoded, the population sizes for the two GA's involved would have been approximately 557 and 2389 instead of 150 and 100, respectively. Schaffer ultimately concludes from intensive empirical testing on a variety of problems, that a population size of 20-30 may be safely used in many situations [Schaffer (1989)].

Goldberg lists both serial and parallel population sizes [Goldberg (1989b)]. For parallel machines, he estimates a population size of "very large to infinite" may be appropriate. This is consistent with other data which showed that performance increased with population size [Booker (1987)]. The large populations on parallel machines, however, are based on the premise that there is no additional cost to adding additional individuals. For serial machines, a constant population of three, is found to be optimal assuming the GA is randomly restarted each time the population converges. Even Goldberg finds this low number surprising.

Both papers report that increasing population sizes does, ultimately, improve performance. Their point, however, is that it may not be worth the cost of such large populations. This basic observation is also echoed in [Jog (1989)].

De Jong's work from 1975 consistently provided good performance both online and offline. Online results are results analyzed during runtime. Offline results refer to the ultimate "best" solution found after a certain period of time. For this reason, his suggested population sizes were selected for experimentation for this project. Population sizes ranging from 25 to 200 were empirically tested on both portions of the current project's genetic algorithm components. The two population sizes thus selected were 100 and 150.

3. Operators. As described in the introduction, a genetic algorithm is comprised of a representation of a problem and the operators which manipulate the data represented. The set of operators considered necessary for a genetic algorithm include the reproduction operator, the crossover operator, and the mutation operator.

a. Reproduction. Reproduction in genetic algorithms involves the selection of individuals from a current population base which will survive into the following generation for further processing. The basic approach is to determine a fitness value for each individual. These fitness values are then summed to provide the fitness of the entire population. Then, surviving chromosomes are selected based upon their contribution to the fitness of the entire population. Several common methods of implementing this ideal are discussed in the literature. It is through this operator that the population should gradually approach the highest levels of fitness, i.e. converge on the global maximum in the search space.

Baker provides an excellent overview of methods generally available in the literature, as well as introducing several new possible choices [Baker (1987)]. The "standard" reproductive operators involve the same basic technique, often called the roulette or spinning wheel method. The spinning wheel method sums the fitness values of the population. Individuals are then mapped one to one onto continuous segments of the real number line. This results in each individual "owning" a segment of the number line equal in proportion to its contribution to the overall population fitness. Then, a random number is generated in the range of the covered number line. The individual whose segment spans that random number is the individual selected for that trial.

The four methods which use this fundamental approach are stochastic sampling with replacement, stochastic sampling without replacement, remainder stochastic sampling with replacement and remainder stochastic sampling without replacement. Baker introduces two alternatives, called remainder stochastic independent sampling and stochastic universal sampling.

Stochastic sampling with replacement assumes that the original assignment of individuals to the number line remain constant between selections. This technique makes it theoretically possible that a single individual could fill all slots in the next generation. *Stochastic sampling with partial replacement* decrements the segment spanned by an individual each time it is selected in a trial. This means that an individual cannot be selected to completely fill the next generation.

The remainder sampling methods break the process into an integral part and a fractional part. The integral portions are used to determine which individuals survive in strict accordance with the proportion provided by the integral part. The remaining slots are then filled according to the fractional portions left from the individuals. *Remainder stochastic sampling with replacement* works the same as stochastic sampling with replacement, except only the fractional parts are considered. *Remainder stochastic sampling without replacement* works the same as stochastic sampling without replacement, except, again, only the fractional parts are considered.

Remainder stochastic independent sampling every individual with a probability of greater than one is selected according to its integer part. The fractional portions are used for selected based upon a random number generated. If the current individual, as the entire population is traversed, has an expected value greater than the random number generated between 0 and 1, it is selected. The process is repeated with as many traversals of the population as required, until all slots in the next generation are filled. This technique could theoretically be infinite.

Most intriguing is *stochastic universal sampling*. This technique involves using an N pointer spinner to select N of the population to survive. Only one "spin" of the spinner is required because on the spinner, there is one pointer for each of the individuals to be chosen to survive. For N slots to be filled, and the fitness of an individual F, the fitness of the population P, each individual should have an expected value, EV = (F/P)*N. This multi-pointer scheme assures that each individual gets at least $\lfloor EV \rfloor$ slots in the new population, but no more than $\lceil EV \rceil$ slots.

Baker points out the various effects of these methods. The point of the current project was to converge as rapidly as possible on local maxima. Therefore, a variation of stochastic sampling with replacement was used. The method employed for the project, however, guaranteed that the fittest individual always survived. Then the remaining slots were filled by stochastic sampling with replacement.

An additional reproductive technique designed to prevent premature convergence if used with the proper crossover methods is the population-elitist selection strategy [Eshelman (1991)]. This technique only replaces parents in a population which are worse than the new offspring created. This technique preserves the superior schema in a population, freeing the crossover operator to be more disruptive than normal.

b. Crossover. Crossover is the primary genetic operator for exploration of the search space. The idea behind crossover is that the surviving individuals in the population, the more fit individuals, exchange genetic material to create new offspring. Hopefully, the new offspring, after receiving this exchanged material, will be even more fit than the parents [Eshelman (1989)]:

> Crossover, like mutation, explores the search space by changing the value of some of the bits in a string. Unlike mutation, however, changes in the chromosome produced by the crossover are constrained to those values that have been shown to be viable in so far as they have survived the selection process. Crossover is, in effect, a method for sharing information between two successful individuals.

Spears discusses the relative roles of crossover and mutation in terms of disruption and construction [Spears (1992)]. In the current project, the desired rapid convergence would favor construction over disruption in operator selection and implementation. Spears comments:

> Clearly the role of crossover is construction, but in this case, crossover provides an advantage over mutation. In terms of disruption, mutation can provide higher levels of disruption and exploration, but at the expense of preserving alleles common to particular positions.....Mutation serves to create random diversity in the population, while crossover serves as an accelerator that promotes emergent behavior from components.

The crossover rate determines the likelihood that a particular chromosome will be involved in a crossover and ultimately determines how many of the next generation were affected by crossover and how many were not. Schaffer reports crossover rates from 0.60 to 0.95. His research indicates the higher range, specifically 0.75 to 0.95 [Schaffer (1989)]. In the current project, based upon Schaffer's comment that "There is evidence that the lowest crossover rates are not associated with best online performance," all of the population was subjected to crossover. Any parents selected were subjected to crossover.

In the literature, there are various forms of crossover operators discussed. These are: one-point (traditional) crossover, two-point crossover, multi-point crossover, segmented crossover, shuffle crossover, uniform crossover, order crossover, cycle crossover, and partially-mapped or PMX crossover. It should be noted that none of these variations were used for the current project. Instead, a unique crossover was used which used dominance weightings to determine crossover applications.

One-point or traditional crossover operates in three stages. First, two parents are randomly selected from the population. Second, a random position is selected. Third, the segments to the right of the randomly selected position are exchanged, possibly creating two new individuals. *Two-point crossover* treats the chromosomes as a ring instead of a string. Two points are selected at random, the segments are exchanged, and two new offspring are created. *Multi-point crossover* also treats the chromosomes as rings. In this case however, an even number of points are selected and exchanges made between corresponding segments of the two parents. *Segmented crossover* is the same as multi-point crossover, except that the number of crossover points varies.

Shuffle crossover is similar to traditional single point crossover. The difference is that it randomly shuffles bit positions in the two strings simultaneously before crossing them, then unshuffles the strings after the segments to the right of the crossover point have been exchanged. This is primarily used when bits in distant positions may be related.

Partially-mapped crossover, *cycle crossover*, and *order crossover*, are all crossovers related to path representation [Michaelewicz (1992)]. All three were introduced for genetic algorithms attacking the Traveling Salesman Problem.

Uniform crossover was introduced [Ackley (1987)], and examined in detail by Syswerda [Syswerda (1989)]. Instead of being segment oriented, it is a bit oriented crossover method. A random mask is generated of the same binary length as the chromosomes. Two parents are selected. Then, the offspring are constructed by using the mask and its inverse. Child one receives the bit value from parent one in positions where a zero occurs and from parent two in positions where a one occurs. The second child is constructed using the inverse of the original mask. Syswerda provides several test cases in which uniform crossover performs better than traditional or two-point crossover.

Eshelman presents a combined strategy using both reproduction and crossover to prevent convergence of a population [Eshelman (1991)]. This strategy, which is called a mating strategy is referred to as *incest prevention*. The population-elitist selection strategy is used for reproduction. Two parents are only mated to produce new offspring if their Hamming distance is above a certain level. This level decreases over the life of the algorithm, being decremented at any point where no parents are accepted into the pool. This crossover also checks new offspring against the old population, and discards duplicates. It does not, however, check for duplicates within the new offspring population.

c. Mutation. The purpose of introducing mutation into a genetic algorithm is to introduce new genetic material into the population and to prevent premature convergence of the chromosomes. The idea, especially in binary encoded genetic algorithms, is extremely simple. A mutation rate is established, perhaps one in a thousand bits. Then, depending on the specific implementation, the correct number of bits are selected from within the population and their values flipped. As a population converges on a maximum, mutation can serve to scatter a few chromosomes back out into the search space. If any values are lost during the numerous crossover operations, mutation can serve to reintroduce those lost values.

Typical suggested mutation rates have been discussed [Schaffer (1989)]. The review of existing research at that time suggested mutation rates of 0.001 to 0.01. Schaffer's own work suggested the rates should be in the range of 0.005-0.01.

Mutation was specifically excluded from the current project. Experimental results showed that mutation did, in fact, serve to slow down convergence as suggested in the literature: "The mutation operator provides a mechanism for reintroducing lost alleles, but does so at the cost of slowing down the learning process [Mauldin (1984)]." De Jong agrees:

> Since the only way of generating new gene values is via mutation, one can be faced with the following dilemma. If the mutation rate is too low, there can be insufficient global sampling to prevent premature convergence to local peaks. However, significantly increasing the rate of mutation can lead to a form of random search that decreases the probability that new individuals will have high performance [De Jong (1988)].

Spears discusses mutation and crossover in terms of their potential to disrupt and construct individuals. In the current project, disruption would be undesirable as it slows convergence and construction would be desirable as it promotes convergence. According to Spears,

> We define two potential roles of any genetic operator, disruption and construction, and consider how well mutation and crossover perform these roles. Our results show that in terms of disruption, mutation is more powerful than crossover, although it lacks crossover's ability to preserve alleles common to individuals. However, in terms of construction, crossover is more powerful than mutation [Spears (1992)].

In the current project, rapid convergence was desirable. Therefore mutation was contraindicated. As mentioned above, this was confirmed by empirical testing as well.

4. Hybrid Genetic Algorithms. Genetic algorithms have been proven successful in a number of different problem search spaces. Their strength, however, is to search over a wide area of the search space and not necessarily to obtain a global optimum. They improve the overall quality of the population without always finding the best or optimal solution. This introduces the concept of hybrid genetic algorithms. A hybrid genetic algorithm uses the GA to find "good" solutions, then passes these solutions on to another program which is superior at exploiting these search areas in a more confined region of the search tree.

> Finally, it is widely recognized that GA's are not well suited to performing finely tuned local search. Like natural genetic algorithms, GA's progress by virtue of changing the distribution of high performance substructures in the overall population, individual structures are not the

> focus of attention. Once the high performance regions of the search space are identified by a GA, it may be useful to invoke a local search routine to optimize members of the final population. [Grefenstette (1987)]

This exploitation of high performing individuals is necessarily problem specific. It will be the problem itself which will determine what the secondary part of the hybrid system may require [Goldberg (1989c)].

There are few articles available describing hybrid genetic algorithms. The project with the most information available is a hybrid system to do automated learning in regards to feature detection [Tamburino (1990), Tamburino (1992), Rizki (1991)]. This system uses a genetic algorithm to perform subset optimization. These subsets are feature sets which are then passed to a neural network feature classifier system. Little information is provided in the series of papers beyond the fitness function and the claim that a "large population of encoded sets is generated" [Tamburino (1992)].

A system has been described involving quadratic assignment problems and which uses a GA in tandem with a simulated annealing program [Huntley (1991)]. This attempt, named SAGA, is of particular interest to the project at hand. The authors note that the computational cost of SAGA could require several days of processing time, and therefore, the SAGA approach is more "greedy" than traditional GA's. One of the techniques used is to combine two parents into a single offspring. The SAGA technique, however, was more complex than that used here. The SAGA crossover operator involved a peculiar variation of the PMX operator which included randomly permuting a subsection of a chromosome.

An economic modelling system has been discussed [Sano (1992)]. This project combined ID3, a neural network, Case-Based Reasoning, a Grossberg Net and a Genetic Algorithm to provide economic predictions. There is insufficient information, however, to determine of what the GA implementation consisted.

One publication claims to be a hybrid genetic algorithm, but does not seem to fit the concept as mentioned by Greffenstette and Goldberg above. An interesting GA is constructed to solve the 3SAT problem in logic [Young (1990)]. Young's idea of a hybrid GA seems to revolve around the fact that the operators on chromosomes are logic-based. There seems to be no portion of the system which would cause it to be considered "hybrid" in the sense indicated above. In Young's project, as with the project at hand, there is no reason to expect that schema will have any relationship to the indication of ultimate convergence on the high performing search spaces.

> ...One point to make is that standard genetic algorithms depend upon the "building block hypothesis" that hear optimal performance can be identified through the juxtaposition of short, low order, high performance schemata. In the SAT problem this hypothesis does not hold in its standard from. The "individuals" used in the algorithms are strings of truth assignments to atomic propositions. These atomic propositions stand in a fixed, but essentially arbitrary, order, which bears no relationship to their associations in clauses. There is no reason in this case to expect that schema ... of short defining length will have greater significance...[Young (1990)].

In the current project, a unique approach has been taken. The total system is classified as a hybrid genetic algorithm. The secondary portion of the program, however, is also used as the evaluation function for the genetic algorithm portion.

Therefore, both the second stage of the hybrid system and the evaluation function are , in fact, the same program. The reasoning behind this multiple use of the evaluation function is simple. At this time, there is no known way of evaluating a word set as to its "value" in a Crozzle, except by attempting to build a Crozzle with that set.

III. THE CROZZLE

A. INTRODUCTION

1. Crozzle Rules. The Crozzle puzzle consists of an empty grid with fifteen columns and ten rows, a word list, which changes monthly, and a set of rules for construction and scoring. The puzzle is published as a contest with a monthly cash prize of $2000A. The rules for the Crozzle are established by the *Australian Women's Weekly* magazine which, in reality, is published monthly. These rules cover both the submission of entries and the construction of legal solutions eligible for entry. Only the construction rules will be discussed here.

A word list is supplied each month. An example word list, from the Crozzle published November, 1991, is shown in Table VI. Letters which interlock in a solution are given various point scores. These point scores are shown in Table V. The point scores for letters have remained the same since October of 1987. Different scoring rules were in effect prior to that time. Only Crozzle solutions since October 1987 through February, 1992, inclusively, will be discussed in this paper.

Table V. LETTER VALUES

a,b,c,d,e,f	2
g,h,i,j,k,l	4
m,n,o,p,q,r	8
s,t,u,v,w,x	16
y	32
z	64

Table VI. SAMPLE LEXICON

crab	reefs	oyster	slipper	seawater
fins	rocks	paddle	snorkel	seaweeds
fish	roses	pistol	soldier	strombid
kelp	shore	prawns	sponges	sunlight
line	snail	ribbon	squirts	ascidians
moon	sting	sharks	textile	barnacles
pipi	tides	shells	trochus	estuarine
reef	tiger	spades	urchins	flatworms
salt	water	squids	waratah	greenweed
sand	waves	triton	baitweed	jellyfish
surf	weeds	turret	bivalves	lifesaver
wind	whelk	whales	breakers	skeletons
algae	anchor	anemone	carapace	strapweed
beach	bailer	chitons	crayfish	sunscreen
cilia	bubble	fishing	crevices	tentacles
clams	bucket	keyhole	cunjevoi	tunicates
claws	castle	lettuce	currents	asteroidea
coast	cliffs	limpets	eelgrass	breakwater
coral	cowrie	lobster	hydrozoa	periwinkle
crabs	cunjee	mussels	littoral	protoplasm
dunes	dumper	neptune	molluscs	underwater
jelly	fronds	octopus	plankton	crustaceans
larva	helmet	pincers	protozoa	echinoderms
mitre	island	planula	scallops	gasteropods
ocean	marine	ripples	scavenge	microscopic
pools	medusa	seaweed	seabirds	beachcombers
prawn	nature	shrimps	seashore	

Each word inserted in the grid, according to the rules, scores ten points. The winning solution, scoring 616 points, is shown in Figure 3 for the word list in Table V. This winning score was constructed by using twenty-three words, for 230 points, and interlocking letters scoring 386 points.

R	O	S	E	S	■	■	P	R	A	W	N	S	■	M
■	■	U	■	A	■	■	[illegible]	■	■	[illegible]	■	T	■	U
M	A	R	I	N	E	■	[illegible]	■	■	[illegible]	■	I	■	S
O	■	F	■	D	■	■	[illegible]	■	■	[illegible]	■	N	■	S
L	■	■	O	■	■	■	O	■	C	■	■	G	■	E
L	■	H	Y	D	R	O	Z	O	A	■	R	■	■	L
U	■	■	S	■	O	■	O	■	S	H	E	L	L	S
S	A	L	T	■	C	O	A	S	T	■	E	■	I	■
C	■	■	E	■	K	■	■	■	L	■	F	I	N	S
S	H	A	R	K	S	■	W	E	E	D	S	■	E	■

Figure 3. Sample Crozzle Solution

Paraphrased, the rules for the Crozzle are as follows:
1) Use only the words in the word list for this month's contest. Each word used in the solution scores 10 points.
2) Words cannot be used more than once in any solution.
3) You cannot run single words together. You must have at least one black square between words which are not interlocked.
4) Letters standing alone have no value. Letters which are interlocked score the appropriate letter values.
5) Words may not stand alone. The finished solution must be a single interlocked block.
6) All entries must be received by the final entry date, approximately 30 days after the puzzle appears [Harris (1993a)].

2. Historical Information. The Crozzle has been operating under the current rules and scoring system since October of 1987. Table VII shows basic information regarding each of these puzzles, through October of 1992.

Table VII. HISTORICAL INFORMATION

Month	HWS	Words in Lex	Words in Solution	Longest Word in Solution	Z Words in Lex?	Z Words in Solution?
Oct87	764	125	30	7	Y	Y
Nov87	810	128	32	7	Y	Y
Dec87	680	112	25	8	Y	N
Feb88	720	115	26	7	Y	Y
Mar88	626	118	24	7	Y	Y
Apr88	836	140	34	7	Y	Y
Jun88	816	140	33	7	Y	Y
Jul88	764	124	29	7	Y	Y
Aug88	696	88	26	7	Y	Y
Sep88	676	107	27	10	Y	Y
Oct88	716	114	26	7	Y	Y
Nov88	630	118	25	8	Y	Y
Feb89	746	114	27	10	Y	Y
Mar89	652	140	20	8	Y	Y
Apr89	768	118	28	7	Y	Y
May89	764	106	29	8	Y	Y
Jun89	760	111	26	6	Y	Y
Jul89	818	126	31	7	Y	Y
Aug89	634	99	25	8	Y	Y
Sep89	616	121	23	7	Y	Y
Oct89	576	140	22	8	Y	N
Nov89	692	123	29	7	Y	Y
Dec89	678	117	25	9	Y	Y
Jan90	612	86	23	7	Y	Y
Feb90	714	127	24	7	Y	Y
Apr90	720	97	25	8	Y	Y
May90	734	122	27	7	Y	Y
Jun90	686	99	28	8	Y	Y
Jul90	626	106	23	8	Y	Y
Aug90	592	113	23	10	Y	N
Sep90	736	141	26	7	Y	Y
Oct90	722	123	30	6	Y	Y
Nov90	652	126	25	8	Y	Y
Dec90	634	101	26	6	Y	N
Feb91	712	114	28	6	Y	Y
Mar91	518	98	23	8	Y	N
Apr91	728	107	29	7	Y	Y
May91	688	111	29	8	Y	Y
Jun91	676	130	24	8	Y	Y
Jul91	710	119	30	7	Y	Y
Aug91	696	118	25	7	Y	Y
Oct91	598	117	21	10	Y	Y
Nov91	616	134	23	8	Y	Y
Jan92	522	124	19	10	Y	Y
Feb92	558	110	22	8	Y	Y

a. "Z" Words. In the forty-five Crozzle contests listed in Table VII, all of contained at least two words in the lexicon with a "Z" in them. As can be seen from Table VI, the interlocking play of two words using a "Z" scores sixty-four points, far more than any other letter. From this observation, one would expect the winning solutions to contain at least one set of interlocking "Z"s. Although this is a common occurrence, it is not universal, however. Of the forty-five Crozzles listed, five do not contain interlocking "Z"s, even though "Z" words were available. From this historical data, it would appear that solutions containing interlocking "Z"s will appear approximately 89% of the time.

Attempts have been made to outscore the winning solutions for the Crozzles containing "Z" words, where the "Z"s were not interlocked. To date, however, no such attempt has exceeded, or even equalled the solution without the interlocked "Z"s.

b. Basic Blocks. *Basic blocks* is the name given to a special word play used appearing in many Crozzle solutions [Harris (1993a), Harris (1993b)]. The fundamental idea is to create a highly interlocked portion of the grid, resulting in a high score. Basic blocks are used in approximately one-third of the winning solutions. An example of a basic block is given in Figure 4, with the letters participating in the basic block shown with double lines around them. This basic block, from the February, 1991 Crozzle, involves eight words, scoring eighty points, and thirteen interlocked letters, scoring 150 points, for a total block score of 230 points. The score for the entire solution was 712 points. Thus, the basic block contributed 32% of the total score. This serves to illustrate the potential of basic blocks in generating high scoring solutions.

More precisely, a basic block is a word play which involves multiple words being placed into the grid at once, no one of which could be removed without causing an illegal solution. As can be seen below, if any of the words involved in the basic block were removed, a situation would occur whereby at least one portion of the basic block contains a word not in the lexicon.

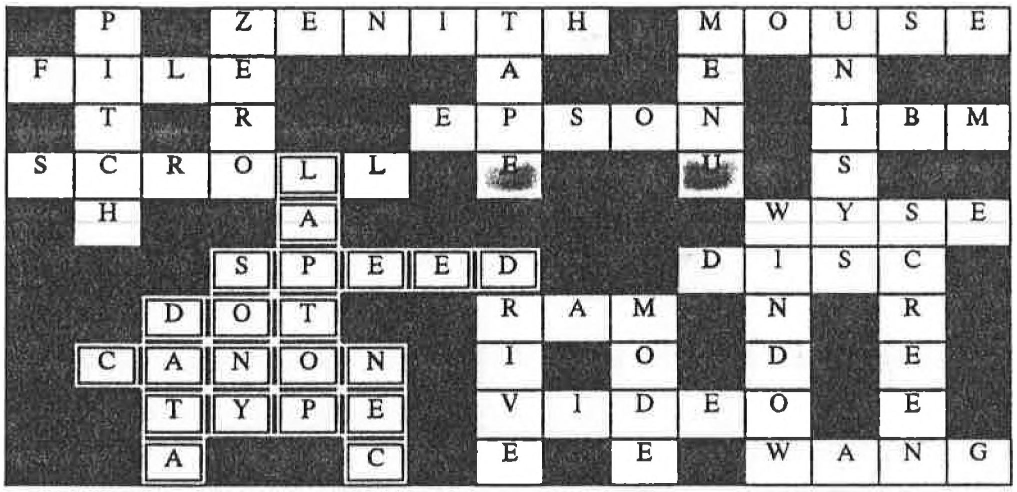

Figure 4. Basic Block

c. <u>Word Lengths</u>. The word lists supplied for each Crozzle contain words with lengths ranging from four to twelve, with occasional words of length three, or longer than twelve characters. The words actually appearing in winning solutions, however, generally do not make use of the longer words. As can be seen from Table VII, the longest word appearing in any of the Crozzle winning solutions, is of length ten. The

Table VIII. SAMPLE BASIC BLOCK SCORING

Letter	Row	Column	Score
s	6	4	32
p	6	5	8
d	7	3	2
o	7	4	8
t	7	5	16
a	8	3	2
n	8	4	8
o	8	5	8
n	8	6	8
t	9	3	16
y	9	4	32
p	9	5	8
e	9	6	2
TOTAL			150

shortest 'longest' word in any one of the published winning solutions is of length 6. The average length of the longest word appearing in the winning solutions is approximately 7.4.

The importance of this observation is that one can generally ignore parts of the supplied lexicon with a reasonable likelihood of not losing any words which will appear in the winning solution. Ignoring all words with a length greater than nine, for example, will only make it impossible to achieve the winning score about 11% of the time. Since longer words require more processing time, due to their greater lengths, the arbitrary elimination of longer words may be an acceptable trade-off for an automated Crozzle solver.

3. Number of Words in Solutions. Winning solutions have used from nineteen to thirty-one words in their solution. The highest number of words used was in the

solution for the April, 1988 Crozzle. The fewest number of words was nineteen, in the January, 1992 Crozzle.

The average number of words appearing in a winning solution is approximately twenty-six. Forty percent of the winning solutions contain more than twenty-six words, with sixty percent containing 26 words or fewer.

B. LITERATURE REVIEW

1. Crossword Puzzles.

a. First Attempts. The first published attempts at crossword puzzle solution were those of Mazlack [Mazlack (1976)]. Mazlack originally tried inserting entire words into puzzle grids, but found the method was not viable. He then used probability considerations to insert letters and construct words in the grids letter by letter. This approach was able to solve some few small puzzles.

b. Static Slot Tables. Smith and Steen [Smith (1981)] are generally credited with developing the first viable method of crossword solution. This attempt proposed a formalized approach called the *static slot table.* A slot table is merely a list of word slots appearing in a given crossword puzzle, along with a flag indicating whether the slot is oriented vertically of horizontally.

With the static slot table, slots are filled in the same order in which they appear in the slot table. Because of this, processing efficiency is dependent upon the order in which the slot table entries appear. This consideration is endemic to the concept and has been considered by [Smith (1981), Berghel (1989), Harris (1990b)].

The slot table is considered static because the slot table is constructed prior to the attempt to solve the puzzle. Therefore, any heuristics applied must be applied without the use of any knowledge which might be determinable at runtime.

As an example of a static slot table, consider the example from [Harris (1992c)]. If one has a totally interlocked 3 by 3 grid for the puzzle, the slot table would appear as in Table IX.

Table IX. STATIC SLOT TABLE

Row	Column	Orientation
1	1	H
1	1	V
2	1	H
1	2	V
3	1	H

Using the slot table shown, the first word would always be inserted at row 1, column 1 in the horizontal slot. Then, an attempt would be made to fill the vertical slot beginning at row 1, column 1, but constrained by the letter inserted in position 1,1. This insertion-constraint procedure continues, with backtracking upon failure, until all slots are filled or until a total traversal of the search tree has taken place.

c. Dynamic Slot Tables. The *dynamic slot table* formalism was first postulated in [Smith (1981)] and implemented in [Harris (1992c)]. This method should perhaps be called a "word oriented" dynamic slot table to distinguish it from the "letter oriented" slot table [Harris (1990b)] .

The word oriented slot table builds a slot table with the same information as a static slot table, with the addition of another column. This added column is updated during runtime. When a word is inserted, the new column contains the number of nodes immediately below each slot in the slot table for each possible word insertion. This, in effect, projects the level of the search tree one additional level by examining the added information. The next slot to be filled is the slot with the lowest positive number in the column. This reduces the number of nodes required to be traversed within the search space by choosing to expand the node with the fewest branches out of it. Again using the example from [Harris (1992a)], Table X shows a typical dynamic slot table, and would result in the slot at 1,2, oriented vertically, to be the next slot to be attempted. Because there are no branches from that node, the entire attempt to insert the original word may be abandoned without losing any solutions.

Table X. DYNAMIC SLOT TABLE

Location	Orientation	Nodes Below
1,1	V	2
2,1	H	9
1,2	V	0
3,1	H	9
1,3	V	2

d. Dynamic Slot Tables for Unconstrained Word Puzzles. The Crozzle is an unconstrained crossword puzzle problem. This means that, at the beginning, no black squares exist, and the black squares are added during the processing of the grid. Because no black squares exist prior to runtime, a static slot table cannot be constructed in advance of runtime.

To attempt a static slot table implementation on the Crozzle, the first step would be to generate a configuration. This configuration would be one of only 2^{150} possible configurations. The generation of these configurations is an NP-Complete problem [Garey (1978)]. For each of these configurations, a static slot table would be constructed and an attempt to solve the constrained problem would follow. The solution to a constrained crossword puzzle is also NP-Complete [Garey (1978)].

The word oriented dynamic slot table is not viable for the Crozzle, either. This approach also constructs word slots prior to runtime. The dynamic portion of the algorithm involves updating the number of branches from each node. But, the slot table must exist prior to runtime. Therefore, again, one of the possible configurations would need to be generated in advance, then a slot table for that configuration constructed, and then an attempt could made at solving the puzzle.

The letter oriented dynamic slot table is an attempt to avoid the problems of the word oriented dynamic slot table. In this approach, the slot table is basically letter oriented. A starting location in the grid is selected and a word inserted. Then, the letters and potential word slots are updated in the slot table. Obviously, the slot table can only be fully constructed at runtime, since the letter positions can only be determined at runtime. This approach, therefore, is called a *run-time dynamic slot table.*

The run-time dynamic slot table is capable of generating all solutions to an unconstrained crossword puzzle, but cannot execute on a non-trivial problem in a reasonably finite time. Therefore, it is necessary, in implementation, to add additional parameters which trim the search space to a more reasonable size. The addition of these parameters, which are effectively search tree pruning heuristics cause an incomplete traversal of the search space, but do so in a reasonably finite time. These parameters are still under investigation. The run-time dynamic slot table is the implementation used for the evaluation function for this project.

2. Crozzle Solver Performance. The performance of various methods utilizing slot table approaches on the Crozzle are mentioned in the literature [Harris (1992a)]. Performance is rated as a percentage of the HWS yielded by the approach. Achievement of 100% of the HWS using this method is rare and unpredictable. The ratings were determined by running each implementation on past Crozzle puzzles with published solutions and comparing the results.

Using a static slot table and a random generation of various puzzle configurations, scores of approximately 60% of the HWS were obtained. Using a static slot table and randomly generating word subsets for each slot to trim the search space again produced scores in the 60% range. By generating word subsets which contained the highest scoring letter available for play in the grid, scores of 70% were obtained.

An approach outlined in [Harris (1992c)] and expanded in [Harris (1993b)] utilizes basic blocks. Using basic blocks as starting points, scores of approximately 80% of the HWS were obtained. The most recent published effort concerns an intelligent backtracking heuristic. This method rates solutions against the highest solution found so far. The further the current solution is below the highest solution determines the number of levels up the search tree to backtrack. This method can yield scores around 90% of the HWS, but has not done so consistently. A variation of this recently tested involves setting the parameters of the backtracking heuristic to produce a semi-admissible heuristic. The heuristic is only semi-admissible, because the distance function is based on historical data, which may not be valid in the current puzzle. This method yields very good scores (90% approximately) very quickly. It explores too much of the search tree thereafter, however, to improve the score in a reasonable time. It is also extremely sensitive to the settings of the initial parameters.

IV. GOALS AND IMPLEMENTATION

A. INTRODUCTION AND MOTIVATION

Attempting to solve a Crozzle puzzle and generate the high score possible is a difficult problem. The difficulty arises from two factors - the grid size and the lexicon size. As the grid size is fixed and cannot be changed under the Crozzle rules, only the size of the lexicon may be altered. As mentioned above, approximately one order of magnitude of the search space can be eliminated with each five words eliminated from the lexicon. This empirical observation makes trimming the size of the lexicon a desirable goal. Even when the lexicon is substantially trimmed, the problem remains intractable for all practical purposes. However, branch and bound techniques applied to a smaller search space can investigate the resultant search space more closely then they can investigate a larger search space.

The goal of the current project is not to solve the Crozzle problem. The goal is to trim the search space of the Crozzle by eliminating words from the lexicon so that other branch and bound programs can more rigorously traverse the search space of the given puzzle. The obvious desired result is that such lexicon trimming will not lose the highest possible score by eliminating words required to generate that score.

The original aim of the GA programs was to produce a proper subset of the original input lexicon through the use of two applications of a GA program. The first program, called GA8, grouped the lexicon in sets of eight words (i.e. chromosomes represented an eight word set). The second program, called GA10, grouped the lexicon in sets of ten words. The output words of the first program are called the GA8 List, the output of the second program is called the GA10 List. The original plan was to combine these two output lists into the Union List. The Union List, in turn was to be processed further by a traditional Crozzle Solver program (CS).

If these lists, either combined or individually, contained all the words of length greater than or equal to six used in the HWS, the attempt would be considered successful. For the purposes of this project, the CS was not actually run on the word lists generated. All the Crozzles used for data already had known solutions and, therefore, known word solution sets. It was assumed that the CS, given the correct word lists, would, indeed, locate the maximum score. This has, in fact, been the case on random samples, but is not guaranteed as the CS implements a heuristic search.

The system designed is considered a Hybrid Genetic Algorithm. Under current definitions, a hybrid genetic algorithm is basically a pre-processor. The output of the genetic algorithm is then further processed by a separate program. In this particular case, the output would be processed by a Crozzle Solver (CS) program.

There are ten instances of each GA8 and GA10 which attempt to process the word sublists. The number of instances chosen was based upon the number of machines available for processing. Each of the 10 available machines processes one sublist and generates an output list. These output lists are then combined, without duplication, to form the GA8 List or the GA10 List, as appropriate. Before these word lists would be input to the CS, all of the 345 words would be added to the lists. As discussed above, there was no attempt to trim the 345 words, as they appeared, from discussions with human Crozzle contestants, to be used opportunistically, and not strategically.

Chromosomes are represented as arrays of integers. The integers represent the index of the word in the word list available for that instance of the GA program being executed. The chromosomes are not binary encoded. To have binary encoded the structures would have required additional operators to test the legality of new genes generated during the crossover and mutation phases. This "legality check" has been the subject of much discussion in the literature. The general reported approach to such legality problems is either to implement much more complex operators or to add a decoder portion to the overall system which performs this function. Since the chromosomes and values were so confined in this project, it was unnecessary to binary encode the chromosomes to provide reasonable performance.

Population sizes varied between GA8 and GA10. In the GA8 programs, population size was 150 individuals. In GA10, the population size was 100. These population sizes were derived from experimentation, with the experimentation bounded by published suggestions [Schaffer (1989)].

The target for the system was to run both GA8 and GA10 in under 12 hours on a series of NeXT workstations. There were 10 such workstations available for use. The code was written in gnu C.

The evaluation function used was a general Crozzle Solver (CS). Given a particular word list, there is no known way of evaluating the quality of the list without trying to construct a Crozzle solution from the list. Analysis of past winning Crozzle solutions give no clear indications as to which words in a particular list will appear in the HWS or in what arrangement. As mentioned above, there is a strong likelihood that the solution will contain interlocked "Z" words, and basic blocks. This information, however, is not sufficient to a priori trim a lexicon. Therefore, to evaluate a word list, that list must be used as input to a CS. This is an expensive evaluation function, in that, given enough iterations for a large population and many generations, could take years to run to completion.

B. DESIRED RESULTS FROM THE PROJECT

The difficulty in this particular problem domain is the size of the search space. The search space is estimated to have approximately 10^{24} solutions, many of which may be trivial. Because of the enormity of the search space, the evaluation of word lists by the CS is relatively expensive. This is a factor not considered in "normal" genetic algorithms, which assume an inexpensive evaluation function. Because the evaluation function must be utilized for each individual in each generation, an expensive evaluation function may make the entire approach untenable. Therefore, if this evaluation function is to be used, fewer individuals and fewer generations can be processed than may be processed in the normal approach.

The need for fewer individuals and fewer generations than is customary forced a re-evaluation of the entire genetic algorithm methodology. Instead of seeking the non-convergence of the population, this project required rapid convergence due to the inability to process large numbers of individuals and generations.

The goal of the project, therefore, was to trim the word lists using a rapidly converging genetic algorithm, without the loss of any words required to solve the Crozzle and achieve the HWS. The non-standard operators and operations required to achieve this rapid convergence concern mutation, reproduction and crossover. Each will be discussed in turn, below.

If the typical Crozzle contains 25 words in its winning solution, it would make sense, it seems, to encode chromosomes representing 25 word lists. In practice, however, this is not feasible. The evaluation of even a 25 word list can take 24 hours on a Sun 630 class machine. Evaluating large numbers of these chromosomes is not a reasonable goal. Due to this expensive evaluation function, smaller word lists must be evaluated. In this project, the decision was to find a series of local maxima using the CS on short word lists. The members contributing to these local maxima are then combined in a search for the global maximum represented by the HWS.

C. DESIGN CONSIDERATIONS

Based upon the expensive evaluation function, rapid convergence was considered a desirable feature in the current project. Most operators and design considerations reported in the literature are intended to prevent rapid convergence. Therefore, the entire approach for the current project needed to be focussed on different aspects of the operators than is customary.

Based upon the literature, several key issues were identified in regards to the convergence issue. Most obvious was the need to eliminate mutation as an operator.

Mutation is a disruptive operator, and serves to prevent premature convergence to local maxima through the re-introduction of rejected genetic material. Empirical tests confirmed that mutation would indeed prevent rapid convergence in this project.

Secondly, the population size needed to be significant enough to allow reasonable sampling of the search space, yet small enough to be processed in a reasonable time. Population sizes as low as 25 as suggested by the literature, were tested. Due to the small number of generations employed, however, these small populations did not sample enough of the search space in the time allowed and were not able to provide satisfactory results. Empirical testing on population sizes from 25 to 200, incremented by 25, upon random Crozzle problems indicated that a population size of 100 for the ten element chromosomes and a population size of 150 for the eight element chromosomes was suitable. Note, this is in conflict with some of the literature which seems to support larger populations for longer chromosomes.

Reproduction, although similar to a steady state reproduction system wherein the entire population is not replaced during each generation, was designed to converge as soon as possible. The most superior individuals were always selected for survival.

Crossover was also strongly affected by the change in design. Instead of creating two offspring from two parents, each pair of parents generated a single offspring. This offspring had only the most valuable genetic material from the pair at each locus.

D. OPERATION

The operation of the GA, like most genetic algorithm programs, is fairly simple. There is an initialization of data, an iterative section of code to process generations, and a termination of the program. Specifically, the GA8 and GA10 programs work as follows:

```
Generate Word Lists
Initialize Data Structures
Generate the Population
While More Generations Remain to Process:
        Eliminate Duplicates
        Score the Chromosomes
        Assess Scoring Penalties
        Save the Best Individuals
        Reproduction Phase
        Crossover Phase
        If Mutation Desired : Apply Mutation
EndWhile
```

The generation of word lists is handled by a program distinct from the GA programs called split.c. The input lexicon is processed by dividing the word list, with duplication, into 10 separate sublists. For each of the ten lists, each word in the lexicon was processed one time. As each word is processed, a random number is generated between zero and three inclusively. If and only if the random number generated is zero, that word is added to the current sublist. The selection of these parameters was intended to take an average lexicon of approximately 120 words and generate sublists each containing approximately thirty words.

Since the sublist generation process is very dependent on the random number generator, Table XI shows a sample of run of the distribution generated on tests with approximately the same number of random numbers generated as required by split.c. The figures shown are for ten sample runs, indicating the minimum and maximum number of the possibilities generated. Ideally, each of the four possible values would appear 375 times (1500/4). This is obviously not the case, however, as can be seen from the table.

Table XI. SAMPLE RANDOM NUMBER DISTRIBUTION

Minimum Appearing	Maximum Appearing
335	417
339	406
344	416
335	417
370	382
339	406
363	392
360	393
335	417
370	382

The initial generation of the population is done by randomly filling each gene with an integer in the interval [0..Number_Of_Words - 1]. There is no attempt to protect against word duplication within a chromosome. The words themselves are maintained in an array Number_of_Words in length.

Once the initialization portions of the programs have been completed, the iterative code is executed once for each generation desired. The number of generations in these experiments was fifteen. This number was based upon run-time considerations. In actuality, nearly the same results would have been obtained with fewer generations. Empirical results show that as few as three generations would have provided a success rate of over 50%. Rarely were more than six generations required to identify the fittest individuals.

The first step in processing a population for a single generation is to attempt to eliminate duplications. This process is called *lazy duplicate elimination* and is not a full duplication elimination mechanism. Each word in each chromosome is checked against the remaining words in the chromosome. If it is a duplicated word, a single attempt is made to replace it with another word from the word sublist available to that instance of the GA program. If another duplicated word is randomly selected, it is placed into the chromosome anyway and no further attempts are made to eliminate it. Any chromosomes containing duplicated words are later penalized after the scoring phase.

After the lazy duplicate elimination occurs, all chromosomes are scored for fitness. The method used to score fitness is unique among the literature. The objective evaluation function used to score the chromosomes is the same program to be used to ultimately process the output lists - the Crozzle Solver (CS). The word list represented by each chromosome is assembled and passed to the CS for evaluation. The CS attempts to maximize the Crozzle score of the word list and returns the highest score generated back to the GA program as a fitness score. This score is recorded for each chromosome.

Reproduction is performed in two phases. The ultimate goal of each instance of the genetic algorithm programs is to converge rapidly to the maximum of a subset of the initial word list. In order to accelerate this process, all individuals obtaining the high score in a population are automatically carried forward to the next step phase. This means that any individual scoring the maximum score in a population during a particular generation always survives for the crossover phase. Once these individuals have been copied to the new population, reproduction continues using the method called *stochastic sampling with replacement*.

The stochastic sampling with replacement is implemented as follows. The entire score for the population is obtained. Until the new population has been filled, a random number is generated in the interval [0..population_sum]. The individual in the population whose segment covers that number on the number line, as explained previously, is selected for the new population. This procedure is listed in the function *LOCATE* in the Appendix.

Once the population has been reproduced, the crossover phase begins. The crossover used for this project is unique. Two individuals are selected randomly for crossover, but only one new offspring is produced. This, again, is done to accelerate convergence to a local maximum. This means that for a population size of pop_size, crossover is performed pop_size times, not pop_size/2 times as is customary. The function *CROSSOVER2* in the Appendix performs this operation. The crossover method used is not similar to any of those discussed in Section II. There is no crossover point per se in Crossover2. Each chromosome is compared in its entirety against the other selected parent.

Experiments were performed on the test data sets using traditional one-point crossover. These experiments were not considered successful, even though one-point crossover outperformed the random selection of words. They did not produce word lists which would allow the CS to produce scores higher than those it could achieve with other heuristic methods. For crossover, two individuals are randomly selected from the population as mentioned. These individuals, although haploid, or single strand, chromosomes, are "bred" as diploid chromosomes using a dominance factor. The dominance factor is called the *Average Potential Word Score* (APWS). From analysis of past Crozzle solutions, it is obvious that the HWS tends to prefer words which have high scoring letters in them. Not all words used in the HWS are those high scoring words, however. If they were, trimming the lexicon would be a simple feat of selecting the top N words from the input word list. APWS is discussed below.

When two chromosomes have been selected, they are compared gene by gene, using the locus as the key for comparison. The new individual receives in position i, the more dominant gene from one of the parents in position i. If new_i, P_{1i}, and P_{2i} are the alleles from the new individual and the two parents, respectively, in position i, APWS is the dominance factor. Index returns the allele containing a dominance weighting. Therefore,

$$\text{NEW}_i = \text{INDEX (MAX (APWS}(P_{1i}), \text{APWS}(P_{2i}))).$$

This assignment operation is performed for each i, where $0 <= i <=$ number_of_genes - 1. The next phase, after the new population has been constructed, is the mutation phase. The purpose of mutation is to prevent premature and rapid convergence to a local maximum. This is diametrically opposed to the goal of this project. Therefore, although mutation operators are available in the program and can be adjusted as desired, mutation was not used in the runs to obtain the data reported below.

E. OVERVIEW

The genetic algorithm implemented here has numerous parts, most of which are merely instances of two separate programs. There are three distinct phases of the GA: GA8, GA10 and the CS. Only the first two phases are of concern in this work. It is irrelevant whether the GA8 or GA10 phase is executed first. Here, it is assumed that the GA8 programs are executed before the GA10 program.

The first step is to process the input lexicon for the Crozzle word list to be trimmed. This input list is always named testlex.in. The separate program, split.c, takes the input word list and distributes it randomly into ten separate sublists, as described above.

Ten instances of GA8 are started, each using one of the 10 sublists generated. For the data provided, each was run for fifteen generations, then terminated. For each instance, the highest score ever for any individual in any generation is determined. All words of length six or greater appearing in any of these high scoring individuals are consolidated in a single list, without duplication. This results in ten GA8 word lists.

These ten lists are then consolidated into a single list, again without duplication. This is the GA8 word list.

Ten new sublists are generated using split.c. Ten instances of GA10 are started, each using one of the sublists. As with the GA8 programs, the end result is a single list of words, each of which appeared in a high scoring individual in at least once instance of a GA10 program. This is the GA10 word list.

Both the GA8 and GA10 word lists are combined to form the Union List. At this point, one of the three lists can be used as input to the CS for an actual attempt at solving the Crozzle under consideration.

The original method proposed was to always use the Union List for input to the CS. The Union List, however, will obviously be as large as the largest of either the GA8 or GA10 word list. In practice it has always been larger than either. Therefore, the processing time required by the CS for the Union List is greater than the processing time for either the GA8 word list or the GA10 word list. The results are presented in the next section and show the effect of using only the GA8 word list, only the GA10 word list, and the Union List.

F. NEW AND CHANGED OPERATORS

1. Introduction. As mentioned, traditional genetic algorithms are very concerned with not allowing rapid convergence of the process to local maxima. This problem has been discussed at length in the literature with numerous proposed solutions. In the current case, however, rapid convergence is not only allowable, it is a desired result. Due to the difficulty of evaluating Crozzle solutions, the evaluation function required is very expensive when applied to large numbers of individuals over a large number of generations.

The operators and general approach of the current project are designed for rapid convergence to local maxima constructed from subsets of the lexicon. The operators primarily affected are reproduction, crossover and mutation.

2. Reproduction. Reproduction in the current project does not completely follow the normal procedure for selection of individuals for a subsequent generation's population pool. In order to encourage rapid convergence of the GAs, the best individuals are forced to survive to the next population.

During the evaluation of individuals in the population, a record is kept of the highest score for any individual. At the beginning of reproduction, the individuals in the population with the highest score reported are automatically copied as is to the population pool. They are not removed from the old population as candidates.

After the best individuals are copied into the new population, the remaining slots for reproduction are selected using stochastic sampling with replacement. This technique assures that the highest scoring individuals always survive to the crossover phase. It was implemented with the explicit purpose of encouraging rapid convergence.

3. Crossover. Crossover is the most unique portion of the current project. Its uniqueness arises from two facets of its implementation. Again, these alterations are specifically designed to construct the strongest individuals possible in the least amount of time. The crossover used is probably nearest that of single point crossover, more due to its simplicity than its concept. Although there have been discussions in the literature of alternate methods, such as multiple point crossovers, PMX mapping crossover, etc., due to the small size of the chromosomes in the current project, these more elaborate methods were not considered necessary. Additionally, none of these crossover methods were designed to encourage rapid convergence of the population.

Traditional single point crossover was attempted experimentally. The results from normal crossover were disappointing. The populations failed to converge within the time/generation constraints applied. Although the results still exceeded those to be expected of random selection of word sets, these experiments did not regularly derive subsets of words which contained the words required to generate the HWS. The resulting word lists allowed the CS to achieve scores approximately equal to those it obtained using other heuristic methods on the full lexicon.

The first unique fact of the crossover method employed was that of generating a single individual from two parents. This means that crossover is employed the same number of times per generation as the population size, double the normal number of crossovers required. This difference is explained and, in fact, is required by the use of the crossover method employed. There is no explicit crossover point in the method used here. The crossover rate used was 1.0.

The recombination method employed is the second unique facet of the operator. Chromosomes are constructed as haploid or single strand chromosomes. When the crossover phase occurs, however, the chromosomes are "mated" as if they were diploid or two strand chromosomes with the use of a dominance factor. The dominance factor, called the Average Potential Word Score (APWS) is based upon analysis of the past Crozzle solutions.

The APWS is a simple formula which relates the length of a word to its scoring potential in the Crozzle. From examination of past Crozzle winning solutions, it was determined that the words most frequently selected for use in the solution also had a high APWS. Unfortunately, not all words used had high a high APWS and not all words with a very high APWS are used. Therefore, the APWS can only be used as a guide to help select potential words.

Given a word of length l, and the letter score values from Table VI, returned by function LV, the APWS is calculated as:

$$(\forall\ i \in (1..l)\ \Sigma\ \mathrm{LV}(i))\ /l$$

For each iteration of the crossover operator, two parents are randomly selected. The parent chromosomes are compared gene by gene, indexed by the locus. The gene with the higher APWS is placed into the corresponding gene of the offspring. In case of a tie, the 'key' value is the first chromosome selected.

For example, assume two chromosomes, C_1, and C_2 each containing three genes. Chromosome C_1 contains the indices for the words: zoo, dig, and bat. Chromosome C_2 contains the indices for the words cat, dog, and fat. Chromosome C_1 was selected randomly before C_2 was selected. The APWS crossover will be applied to the following pairs, by position: (zoo, cat), (dig, dog) and (bat, fat). Table XII shows the resulting

Table XII. EXAMPLE APWS CROSSOVER

C_{1i} Word	C_{1i} APWS	C_{2i} Word	C_{2i} APWS	Offspring Word
zoo	26.66	cat	6.66	zoo
dig	3.33	dog	4.66	dog
bat	6.66	fat	6.66	bat

offspring by position. Notice that, in the third row, "bat" was chosen over "fat" merely because it was in the parent selected first.

4. Mutation. Mutation is used in genetic algorithms to prevent convergence to local maxima by introducing new genetic material, or re-introducing lost genetic material. The mutation operator is applied according to a mutation rate parameter and affects only a small portion of the population at any given time. The idea behind mutation is to randomly and occasionally take a chromosome and move it from its

current data point to a new data point by altering one or more pieces of information in the chromosome. This does two things. It allows a converged chromosome to be moved to a new portion of the search space, thus expanding the search area. It also provides the opportunity to re-introduce any genetic information which may have been lost due to crossover or previous mutations.

In the current project, the desire was to encourage rapid convergence. Therefore, the use of a mutation operator was contrary to the goals. Although the mutation operator was present in the code, it was not used for the data runs discussed herein. When sample runs were made with the mutation operator set at approximately 0.001 mutations, convergence was, indeed, adversely affected, and the programs were not able to converge within the time and generation restrictions applied.

V. RESULTS AND DISCUSSION

A. RESULTS

In order to test the efficacy of the implementation described, ten Crozzle puzzles were selected at random. Each Crozzle word list was used as input to both GA8, and GA10. The output word sublists were compiled and compared to the words in the HWS for each Crozzle.

Table XIII shows the complete sub-lexicons generated after trimming the words of lengths greater than five. This represents the total number of words which would be presented to the CS in the second stage of the hybrid system. The totals include the 345 words as well. On average, both the GA8 and GA10 programs trimmed approximately 30% of the overall lexicon. As discussed in Section I, the fact that this also represents about thirty words eliminated as well, means that approximately six orders of magnitude have been eliminated from the search space. This allows the parameters to the CS portion of the hybrid algorithm to search the remaining space more closely, hopefully, resulting in higher ultimate scores found.

Table XIV shows the performance of the GA's as compared to purely random selection. The expected words found are calculated according to the formula below. If one assumes that, for example, there were 100 words in the lexicon and twenty of those words appeared in the HWS, and 10 words are selected randomly from the lexicon, that two of those ten words would appear in the HWS word list. If W = the number of words selected at random, S = the number of words in the HWS and L = the number

Table XIII. TRIMMING OF THE TOTAL INPUT LEXICON

	Original Words	B GA8 Total Words	C (B/A)*100 (% left)	D GA10 Total Words	E (D/A)*100 (% left)
Jan92	120	73	60.83	76	63.33
Feb92	104	62	59.62	70	67.31
Jul91	107	77	71.96	81	75.70
Apr91	104	75	72.12	77	74.04
Dec90	100	78	78.00	74	74.00
Feb90	124	82	66.13	80	64.52
Aug89	96	73	76.04	67	69.79
Oct89	121	77	63.63	73	60.33
Feb88	97	70	72.16	75	77.32
Oct90	102	70	68.63	80	78.43
AVERAGE	107.50	73.70	68.91	75.30	70.05

of words in the lexicon, the formula for expected HWS words, EX, found is

$$EX = W * (S/L)$$

The number of words with lengths greater than five are shown in parenthesis for each month's HWS word list. Obviously, those entries which show the same number of HWS words found as there were in the HWS were successful in trimming the lexicon without losing any of the required words to generate the HWS. Those which did not succeed are marked with an asterisk in the table.

As can be seen, fourteen of the twenty individual runs did generate all the words in the HWS. When the Union List of each of the ten Crozzles test runs is checked, however, nine of the ten, or 90%, of them contained all the words in the HWS word list. Only the January, 1992 Crozzle failed using the Union List. Although the generation of the Union List was the original goal of the project, the success of the individual GA8 and GA10 programs changed the focus. It had not been anticipated that the GA8 and GA10 programs would prove so successful on their own. Combining the GA8 and GA10 word lists, generally resulted in a Union List approximately ten words larger than either sublist. This means that the Union List had roughly two orders of magnitude more search space to evaluate. Therefore, the results of current efforts, discussed in the proof of concept section below, are based upon using either GA8 or GA10 word lists and not the Union List. The Union List, when it is smaller than normal, has been used for the later efforts at Crozzle solution. None of these results, however, have been submitted to the contest to date. Generally, the runtime is too long to meet the Crozzle submission deadlines.

Table XV shows the results of trimming the lexicon without the inclusion of the 345 words. This table, therefore, directly indicates the effect of the GA8 and GA10 programs on the words of interest. Again, those runs which were considered individually unsuccessful are marked with an asterisk. The importance of this table is the relative trimming of the successful and unsuccessful runs.

Table XIV. PERFORMANCE OF GA VS RANDOM SELECTION

	Expected GA8	Actual GA8	Expected GA10	Actual GA10
Jan92	5.77	* 10 / (11)	6.11	* 10 / (11)
Feb92	4.81	* 9 / (10)	5.80	10 / (10)
Jul91	1.52	3 / (3)	1.72	3 / (3)
Apr91	2.36	* 4 / (5)	2.55	5 / (5)
Dec90	3.60	6 / (6)	3.16	* 4 / (6)
Feb90	3.80	8 / (8)	3.60	* 5 / (8)
Aug89	5.00	8 / (8)	4.38	8 / (8)
Oct89	4.26	8 / (8)	3.91	8 / (8)
Feb88	3.11	6 / (6)	3.64	6 / (6)
Oct90	1.75	4 / (4)	2.46	4 / (4)
AVG.	3.60	6.70 / (6.90) 97.10%	3.73	6.30/(6.90) 91.30%

With GA8, the three runs which failed to retain the words in the HWS, left an average of 49.31% of the longer words in the lexicon. Those GA8 runs which were successful left an average of 52.86% of the words. The unsuccessful GA10 runs left an average of 51.05% of the longer words in the lexicon. The successful Ga10 runs left an average of 56.03% of the longer words. This seems to indicate that those runs which are over-zealous in the trimming of the lexicon are more likely to fail than those which are more conservative. It is believed that this over-trimming is a result of using the APWS weighting during crossover, but it has not been so established at this point.

Another point of note concerning the failed attempts of the single GA8 and GA10 runs involves the number of 345 words available. It appears that Crozzle word lists with too few 345 words are not generally solvable by the GA programs. When there are fewer than approximately twenty-five of the 345 words to use, the GA programs do not seem to succeed as frequently. The cause of this phenomenon is not yet understood. The January 1992 Crozzle is an example. In this puzzle, the GA8, the GA10 and the Union List were all failures. This lexicon contained only twenty of the 345 words.

Table XV. TRIMMING LONGER WORDS

	A Original Long Words	B GA8 Total	C (B/A)*100 (% left)	D GA10 Total	E (D/A) * 100 (% left)
Jan92	99	52 *	52.52	55	55.55 *
Feb92	81	39 *	48.14	47	58.02
Jul91	61	31	50.82	35	57.38
Apr91	55	26 *	47.27	28	50.90
Dec90	55	33	60.00	29	52.72 *
Feb90	80	38	47.50	36	45.00 *
Aug89	62	39	62.90	34	54.84
Oct89	94	50	53.19	46	48.94
Feb88	56	29	51.79	34	60.71
Oct90	57	25	43.85	35	61.40
AVG.	70.43	36.86	52.34	37.71	53.54

Due to the fact that random numbers appear so frequently in a genetic algorithm, it is reasonable to question whether the programs presented here could repeatedly succeed on these problems. In order to investigate this question, one of the 10 trial Crozzles was selected at random and subjected to repeated runs. Tables XVI and XVII show the performance of the GA when repeatedly applied to the same Crozzle puzzle. The February 1992 word list was the one randomly selected for testing. Each of GA10 and

GA8 were run five times on the given word list for that month. There were nine words of the proper lengths in the HWS.

As can be seen, each of GA8 and GA10 succeeded four of the five times. Once again, the failed attempts trimmed more of the lexicon than the succeeding attempts, on average. The successful GA8 runs trimmed an average of 49.99% of the lexicon. The failed attempt trimmed 60%. The GA10 programs which succeeded trimmed an average of 54.75% of the words. The failed attempt trimmed 57% of the words.

Table XVI. REPEATED GA10 SAMPLE RUNS

Words Generated	HWS Words Found	% of Long Words Eliminated
39	9	52
40	9	51
42	9	48
32	8	60
41	9	49

Table XVII. REPEATED GA8 SAMPLE RUNS

Words Generated	HWS Words Found	% of Long Words Eliminated
32	9	60
37	9	53
35	9	57
41	9	49
35	8	57

B. PROOF OF CONCEPT

All of the trial runs were performed on Crozzles with known winning solutions. In order to begin testing the concepts implemented more rigorously, attempts have been made to solve current Crozzle puzzles, about which no information is available other than the lexicon. The trial runs assumed that, given the correct word list, the CS would indeed find the maximum solution. This was assumed since the CS and its efficiency were not part of the current research. In reality, however, this becomes a viable concern. Some of the actual "online" performances are reported here. It should be recalled that, prior to the GA lexicon trimming efforts, the CS was normally able to score within 80% or so of the HWS regularly and as high as 90% on an inconsistent basis. On current Crozzles being published, the CS is now able to score consistently above 95% of the HWS, although no "victories" can as yet be claimed.

For the July 1992 Crozzle, due to a shortage of time before the submission date, a decision was made to arbitrarily exclude words of length 8 and above. The GA programs were then run on the remaining lexicon. The resulting solution submitted was 4 points below the HWS score. The difference in the submitted solution and the winning solution was a 4 point intersection which the HWS made from an 8 letter word. All other words in the solution had been found by the GA programs. When the GA programs were re-run after the fact, including the longer words, all words in the HWS were found! It was not determined whether the CS could have taken that list and found the HWS, however. Even with the word missing, the hybrid programs scored 99.36% of the HWS!

For the August 1992 Crozzle, the GA programs did not find the correct word set, missing one word even when the Union List was considered. This lexicon had only 15 of the 345 words, and led to the observation that the failures to date had all involved Crozzles with a small number of these words.

For the September 1992 Crozzle, the GA programs found all the words in the HWS. The CS, however, was not able to find the winning solution in the allowed time.

The October 1992 Crozzle, the GA's failed to find one of the words in the HWS. Even so, the word list which the GA's did provide allowed the CS to obtain 98.4% of the winning score.

For the November 1992 contest, the GA hybrid programs scored 97.6% of the HWS. The submitted score was mentioned in the magazine when the solution was published. For the December 1992 contest, they scored 96.9% of the HWS.

less severe than the APWS needs to be developed. Mutation, of course, could remedy this problem, theoretically. From empirical tests, however, it would not be able to do so in the short number of generations used. Therefore, additional research needs to be done to investigate how severe the crossover weighting function should actually be in order to encourage convergence without losing important genetic material which does not score well in the weighting.

The concept of basic blocks seems to provide a new avenue with which to attack the Crozzle. There is a very efficient implementation of a CS which generates basic blocks [Harris (1993b)]. However, there are so many basic blocks generated, that it becomes a new problem to select the proper one for seeding the initial state of the solution attempt. A genetic algorithms may be appropriate for the exploration of this problem as well.

One of the most troublesome aspects of the hybrid system is the failure of the CS to locate the HWS even when the GA's provide the proper word lists. This problem is related to the large number of variable search parameters in the CS program. The possibility of using a genetic algorithm to fine tune these parameters for each contest is under investigation.

The January 1993 contest solution was hampered by excessive machine down-time. The GA's located all the words in the HWS, but there was insufficient time to process the list. The CS did, however, locate a solution scoring 95.2% of the HWS in the time available.

It can be clearly seen from these figures that, even when the GA portion of the hybrid fails to isolate all the words used in the HWS, it still trims the search space sufficiently to improve the overall performance of the CS.

C. FUTURE RESEARCH

Several issues remain to be resolved concerning the rapid convergence of genetic algorithms. First, it is not clear how to determine the desirable number of generations needed to define "rapid" for a particular problem. In the current project, 15 generations was chosen based upon time constraints. Most of the GA runs however, had produced the superior individual towards which the population converged by the third generation. Only rarely did the most superior individual emerge after the sixth generation.

It is unclear at this point why the shortage of 345 words adversely affects the performance of the GA programs. Even when, in these cases, each GA is given ALL of the 345 words in its sublist, performance did not meet expectations. This phenomenon needs to be investigated further.

The use of the APWS as a crossover weighting proved to be highly successful. In general, however, when the GA programs miss a word, it is consistently a word with a low APWS. Therefore, to improve overall performance, it seems that some weighting

VI. CONCLUSION

The primary emphasis in the literature pertaining to genetic algorithms concerns operators that discourage rapid convergence in the aims of avoiding local maxima. The problem presented herein, the Crozzle, is not suitable for such approaches.

The Crozzle is an NP-Complete problem which has no known inexpensive method of evaluating the fitness of individuals. Therefore, the traditional method of discouraging convergence makes the problem too lengthy to attempt to solve. An alternative approach has been presented. First, the goal of the project was not to directly solve the Crozzle problem, but to design a hybrid system which used a genetic algorithm to trim the search space for a general Crozzle Solver. In order to reduce the search space encountered by the Crozzle Solver, the genetic algorithms were used to trim the lexicon by finding local maxima based upon subsets of the overall lexicon. These local maxima producing word sets were then recombined for input to the Crozzle Solver.

The general approach of the hybrid genetic algorithm has been shown to be successful. The smaller word lists from the genetic algorithms were empirically tested on a random sampling of problems and each shown to be 70% effective in meeting the desired goals. The union of these lists was empirically shown to be 90% successful in meeting the desired goals.

The success of the project indicates that further research may be justified in developing genetic algorithms which converge rapidly. The indications are that alterations in the reproduction, crossover, and mutation operators are required for this to be successfully accomplished. Problem specific knowledge can be used to affect the results of the crossover operator to encourage suitable convergence. The alterations in the mutation and reproduction operators required no problem specific information. The mutation operator was not required for convergence and, in fact, proved a hinderance, as might be expected. The reproduction operator was only altered by enforcing the survival of the fittest individuals. Beyond that, it merely used stochastic sampling with replacement. Therefore, the indications are that the crossover weighting function would prove to be the most serious obstacle to obtaining suitable results on similar problems.

APPENDIX

SOURCE CODE

```
                    /* Split.c */
/* This program was used for the trials runs to divide the lexicon into sub-lexicons.
These sub-lexicons were used as input wordlists to the various instances of the GA8 and
GA10 program. This program has been replaced with a version providing a more
equitable distribution. The new program is being used in current efforts */

#include <stdio.h>
#include <time.h>
#include <ctype.h>
#include <string.h>

#define true 1
#define false 0
#define MAX_WORDS 150
#define LEX_DIVISOR 4
#define MAX_WORD_LENGTH 15

typedef char (string [15]);
typedef string (sss [MAX_WORDS+1]);

int used [MAX_WORDS + 1];
sss word_list;
int number_of_words, x, y, z, loop, a, b, c;
FILE *inlex, *outlex;



main ()
{

srandom(time(NULL));
for (x=1;x<=MAX_WORDS;x++)
   {
    used[x] = 0;
   }
number_of_words = 1;
if ((inlex = fopen("testlex.in","rt")) == NULL)
   {printf("file error on testlex.in\n");
    exit(0);
   }

while ((fscanf(inlex,"%s",word_list[number_of_words]) != EOF ))
{
  number_of_words++;
}
fclose(inlex);
number_of_words--;
printf("Read %d words \n",number_of_words);
for (x=1;x<=10;x++)
{
  switch (x)
   {
   case 1: outlex = fopen("next1/sublex.in","wt");break;
   case 2: outlex = fopen("next2/sublex.in","wt");break;
   case 3: outlex = fopen("next3/sublex.in","wt");break;
   case 4: outlex = fopen("next4/sublex.in","wt");break;
   case 5: outlex = fopen("next5/sublex.in","wt");break;
   case 6: outlex = fopen("next6/sublex.in","wt");break;
   case 7: outlex = fopen("next7/sublex.in","wt");break;
   case 8: outlex = fopen("next8/sublex.in","wt");break;
   case 9: outlex = fopen("next9/sublex.in","wt");break;
   case 10: outlex = fopen("next10/sublex.in","wt");break;
   }
printf("processing file # %d\n",x);
  for (loop=1;loop<=number_of_words;loop++)
    {
     a = random() % LEX_DIVISOR ;
    if (a==0) {
                fprintf(outlex,"%s\n",word_list[loop]);
                used[loop]++;
              }
    }

fclose(outlex);
```

```
} /* for x */
for (x=1;x<=number_of_words;x++)
{
  if (used[x] == 0)
   {printf("Unused: %s\n",word_list[x]);
    y = random () % 10 ;
    y++;

  switch (y)
   {
   case 1: outlex = fopen("next1/sublex.in","at");break;
   case 2: outlex = fopen("next2/sublex.in","at");break;
   case 3: outlex = fopen("next3/sublex.in","at");break;
   case 4: outlex = fopen("next4/sublex.in","at");break;
   case 5: outlex = fopen("next5/sublex.in","at");break;
   case 6: outlex = fopen("next6/sublex.in","at");break;
   case 7: outlex = fopen("next7/sublex.in","at");break;
   case 8: outlex = fopen("next8/sublex.in","at");break;
   case 9: outlex = fopen("next9/sublex.in","at");break;
   case 10: outlex = fopen("next10/sublex.in","at");break;
   }
  fprintf(outlex,"%s\n",word_list[x]);
 printf("Adding  %s  to list %d\n",word_list[x],y);
  used[x]++;
  fclose(outlex);
} /* if */
}  /* for x */


}  /* main */
```

```
/* GA.C */
```

/* This is the source code for the GA8 and GA10 programs. For GA8 instances, max_chrom is set to 8, and max_pop is set to 150. For GA10 instances, max_chrom is set to 10, and max_pop is set to 100. The code for mutation, which was not used in the trial runs, and the code for traditional single-point corssover, also not used in the trial runs, is included. The code used for the evaluation function is not included. That code is copyrighted by G. Harris, and is considered as a 'black box' to the current project */

```
/* ===================================*/
#include <stdio.h>
/* ===================================*/
#define begin {
#define end }
#define LEX_DIVISOR 3
#define MUTATE_ON 0
#define MUTATE_AT 2
#define MAXIMUM_WORDS  150
#define DUPLICATE_PENALTY  50
#define max_chrom 8
#define GENS 15
#define max_pop  150
#define sw_copy 15
#define IFT 4
#define ILAST 4
#define XLEN 10
#define YLEN 10


#include <ctype.h>
#include <string.h>
#define true 1
#define false 0
#define begin {
#define end }



/* ===================================*/
```

```
void GETWORDS (sss word_list, int *number_of_words)
 /* ====================================*/
begin
  FILE *textfile;
  int x,zipper;
  if ((textfile = fopen("sublex.in","rt")) == NULL)
     begin
       printf("Error opening text file for reading\n");
       exit(0);
     end
  x = 1;
  while ((fscanf(textfile,"%s",word_list[x]) != EOF))
  begin
/*
zipper = random() % LEX_DIVISOR;
printf("%s   %d   %d \n",word_list[x],x,zipper);
if (zipper == 0) begin
        x++;
     end
*/
x++;
  end
  fclose(textfile);
  *number_of_words = --x;
end;

 /* ====================================*/
void INITPOOL(  struct zz *p)
 /* ====================================*/
begin
int x,y,z;
for (x=1;x<=max_pop;x++)
  begin
    for (y=1;y<=max_chrom;y++)
      begin
         p[x].chrom[y] = random() % number_of_words + 1;
        words_used[p[x].chrom[y]]++;
      end;
  end;
end; /* initpool */

 /* ====================================*/
void PRINTPOOL( struct zz *p , sss word_list)
```

```
/* ====================================*/
begin
 int x,y;
 for (x=1;x<=max_pop;x++)
 begin
   printf("CHROMO: %d  GENES:",x);
   for (y=1;y<=max_chrom;y++)
     begin
       printf(" %d  %s",p[x].chrom[y],word_list[p[x].chrom[y]]);
      end
    printf(" SCORE: %d  \n",p[x].score);
  end
end

 /* ====================================*/
void PRINT_SCORES (struct zz *p)
/* ====================================*/
begin
int x,y;
 for (x=1;x<=max_pop;x++)
   begin
    printf(" %d   %d \n",x,p[x].score);
   end
end

 /* ====================================*/
void ASSESS_PENALTIES(struct zz *p)
/* ====================================*/
begin
int x,y,z;
for (x=1;x<=max_pop;x++)
  begin
    for (y=1;y<= (max_chrom-1);y++)
       begin
         for (z=(y+1);z<=max_chrom;z++)
           begin
             if (p[x].chrom[y] == p[x].chrom[z])
               begin
                 p[x].score = p[x].score - DUPLICATE_PENALTY;
               end
           end
         end
   if (p[x].score < 0) begin p[x].score = 0; end
  end
end


void MUTATE(struct zz *old_pool, sss word_list)
begin
  int w_gene,w_chrom,w_word;
  w_gene = (random() % max_pop) +1;
  w_chrom = (random() % max_chrom)+1;
  w_word = (random() % number_of_words) + 1;
/*
printf("            current value: %d\n",old_pool[w_gene].chrom[w_chrom]);
*/
  old_pool[w_gene].chrom[w_chrom] = w_word;
/*
printf("            replacing %d  %d  with %d\n",w_gene,w_chrom,w_word);
*/
end


double SCORE_WORD(int POS)
begin
  int x,y,z,sum, len;
 sum = 0;
len = strlen(word_list[POS]);

 for (x=0;x<len;x++)
  begin
     sum = sum + values[word_list[POS][x]-'a'];
  end
return ((double) sum / (double) len);
end

void CROSS2 (struct zz *old_pool, struct zz *new_pool)
begin
  int toss;
double sc1,sc2;
   int x,y,y1,y2,z,b,loop;
for (x=1;x<=max_pop;x++)
  begin
  y1 = random() % max_pop +1;
  y2 = random() % max_pop + 1;
  while (y1==y2) begin y2 = random() % max_pop +1; end
```

```
for (y=1;y< =max_chrom;y++)
   begin
     sc1 = SCORE_WORD(old_pool[y1].chrom[y]);
     sc2 = SCORE_WORD(old_pool[y2].chrom[y]);
    if (sc1>sc2)
       begin  new_pool[x].chrom[y] = old_pool[y1].chrom[y];
       end
    else begin new_pool[x].chrom[y] = old_pool[y2].chrom[y];
         end
end /* for y */
end /* for x */
end /*function */

  /* ====================================*/
void CROSSOVER (struct zz *old_pool , struct zz *new_pool)
  /* ====================================*/
 begin
  int x,y,y1,y2,z,at,b;
   for (x=1;(x< =max_pop / 2) ;x++)
   begin
     y1 = random() % max_pop + 1;
     y2 = random () % max_pop +1;
     while (y1 == y2)
     begin
       y2 = random() % max_pop +1;
     end
     at = random() % max_chrom;
     while (at == 0)
       begin
          at = random() % max_chrom;
       end;
     /* printf("y1 :%d   y2 :%d  at:%d\n",y1,y2,at); */
     for (z=1;z< =at;z++) begin
         new_pool[x].chrom[z] = old_pool[y1].chrom[z];
         new_pool[x+(max_pop / 2)].chrom[z] = old_pool[y2].chrom[z];
         end
     for (z=at+1;z< =max_chrom;z++)
     begin
         new_pool[x].chrom[z]=old_pool[y2].chrom[z];
         new_pool[x+(max_pop) / 2].chrom[z] = old_pool[y1].chrom[z];
      end
    end
 end

 void GETSCORE ()
 begin
     step_one();
     for(pivot=ift;pivot<=ilast;++pivot) insert_first_word(pivot);

 end

 void SCORE_CHROMOSOMES (struct zz *old_pool)
 begin
   int x,y;
FILE *outfile;
   for (x=1;x<=max_pop;x++)
   begin
     printf("     scoring chromosome # %d\n",x);
     if ((outfile = fopen("crozzlewords2.in","wt")) == NULL)
    {printf("cannot open output file.\n");}
     for (y=1;y<=max_chrom;y++)
     begin
       fprintf(outfile,"%s\n",word_list[old_pool[x].chrom[y]]);
     end
fprintf(outfile,"\n");
     fflush(outfile);
  fclose(outfile);
     GETSCORE();
     old_pool[x].score = hscore;
   end /* for x = 1 to max_pop */
 end



 /* ====================================*/
 void LOCATE (int here , int  y,struct zz *old_pool, struct zz *new_pool)
 /* ====================================*/
 begin
 int x,loop;
 x = 1;
 while (y>0)
  begin
    y = y - old_pool[x++].score;
  end
  --x;
  new_pool[here].score = old_pool[x].score;
  new_pool[here].used = old_pool[x].used;
  for (loop=0;loop<=max_chrom;loop++)
 {new_pool[here].chrom[loop] = old_pool[x].chrom[loop];}

 end

 /* =================================*/
 void BLABBER (int y7, int save_pos,int total_scores,
              int highest_ever, int worst_ever,
              int original_high)
 /* =================================*/
 begin
 printf("==============================\n");
 printf("highest score was %d found at position: %d \n",y7,save_pos);
 printf("Total Scores for this generation was: %d\n",total_scores);
 printf("Highest Score Ever: %d   Worst Score ever:  %d \n",
                                  highest_ever,worst_ever);
 printf("Highest score in original pool was %d\n",original_high);
end
/* =================================*/
void FINISH_UP( struct zz *old_pool,
               int y7, int worst_ever, int highest_ever,
               int original_high)
/* =================================*/
begin
FILE *outfile;
 outfile = fopen("ga.out","at");
fprintf(outfile,"stop time: %d \n",(unsigned int) time(NULL));
fprintf(outfile,"Words in lex: %d\n",number_of_words);
fprintf(outfile,"Generations: %d  Population: %d  Chromosomes: %d\n",
         GENS,max_pop,max_chrom);
fprintf(outfile,"Last highest: %d  Highest Ever: %d \n",y7,highest_ever);
fprintf(outfile,"Worst ever: %d  Original high: %d\n",worst_ever,original_high);
fflush(outfile);
fclose(outfile);
printf("\n\n");
/*  PRINTPOOL(old_pool,word_list);  */
 printf("Last highest score was %d\n",y7);
 printf("Highest Score Ever: %d   Worst Score ever:  %d \n",
                                  highest_ever,worst_ever);
 printf("Highest score in original pool was %d\n",original_high);
 printf("FINISHED PROCESSING\n");
  end


 int IN(int x , int y , struct zz *old_pool)
 begin
    int a,b,c;
     b = false;
     for (a=(y+1);a<=max_chrom;a++)
     begin
     if ( old_pool[x].chrom[y] == old_pool[x].chrom[a])  begin
                               b=true;
                       end
                                           end
 return(b);
 end

 void ELIM_DUPS(struct zz *old_pool)
 begin
   int x,y,z,loop;
   for (x=1;x<=max_pop;x++)
 begin
       for (y=1;y<max_chrom;y++)
          begin
            if ((z=IN(x,y,old_pool))==true) begin
                 old_pool[x].chrom[y] = random() % number_of_words +1 ;
               end
         end
 end
 end




 void COPY_POOL( struct zz *old_pool, struct zz *new_pool)
 begin
 int x,y;
   for (x=1;x<=max_pop;x++)
     begin
       old_pool[x].score = new_pool[x].score;
       old_pool[x].used = new_pool[x].used;
       for (y=0;y<=max_chrom;y++)
           {old_pool[x].chrom[y] = new_pool[x].chrom[y];
            if (old_pool[x].chrom[y] > number_of_words) {
             printf("copying error in COPY_POOL: %d \n",old_pool[x].chrom[y]);
 }
 }
    end
 end



  /* ===================================*/
 main ()
  /* ===================================*/
 begin
 int x,y,loop;
 printf("Program begins\n");
 values[0] = values[1] = values[2] = values[3] = values[4] = 2;values[5]=2;
 values[6]=values[7]=values[8]=values[9]=values[10]=values[11]=4;
 values[12]=values[13]=values[14]=values[15]=values[16]=values[17]=8;
 values[18]=values[19]=values[20]=values[21]=values[22]=values[23]=16;
 values[24]=32;values[25]=64;

 srandom ((unsigned int) time(NULL));
 printf("Reading Words \n");
 GEN_NUM = 1;
 GETWORDS(word_list,&number_of_words);
 for (loop=1;loop<=number_of_words;loop++)
     {  words_used[loop] = 0; }

 printf("Words Used: %d\n",number_of_words);
 printf("initializing pool\n");
 INITPOOL(old_pool);
 highest_ever = 0;
 worst_ever = 1000;
 outfile = fopen("ga.out","wt");
 for (loop=1;loop<=number_of_words;loop++)
    {
       if (words_used[loop] == 0)
         { fprintf(outfile,"not used: %s\n",word_list[loop]);
            printf("not used: %s\n",word_list[loop]);
         }
    }
 /*
 for (loop=1;loop<=number_of_words;loop++)
   begin
```

```
      fprintf(outfile," %s\n",word_list[loop]);
    end
*/
fprintf(outfile,"Genes: %d  \nPopulation: %d\n",max_chrom,max_pop);
fprintf(outfile,"Number of words in lex: %d\n",number_of_words);
fprintf(outfile,"Lex Divisor: %d\n",LEX_DIVISOR);
fprintf(outfile," start: %d\n",(unsigned int) time(NULL));
fflush(outfile);
fclose(outfile);
while (GEN_NUM <= GENS)
begin
  outfile=fopen("ga.out","at");
  fprintf(outfile,"=======================\n    Gen    %d    of    %d
\n=================",
          GEN_NUM,GENS);
  fflush(outfile);
fclose(outfile);
  printf("Generation Number: %d of %d generations requested\n",GEN_NUM,GENS);
ELIM_DUPS(old_pool);
  SCORE_CHROMOSOMES(old_pool);
  ASSESS_PENALTIES(old_pool);
/* PRINTPOOL(old_pool,word_list);  */
  /* see how things look with the first generation */
  if (GEN_NUM == 1)
  begin
    y = 0;
    for (x=1;x<=max_pop;x++)
     begin
      if (old_pool[x].score > y) {y = old_pool[x].score; }
     end
     original_high = y;
   end  /* if First Generation */
   total_scores = 0;
  for (x=1;x<=max_pop;x++)
  begin
   total_scores += old_pool[x].score;
  end
  y7 = 0;

   for (x=1;x<=max_pop;x++)
 begin
   if (old_pool[x].score < worst_ever)
      begin worst_ever = old_pool[x].score; end
   if (old_pool[x].score > y7)
   begin
     y7 = old_pool[x].score;
     save_pos = x;
     end
  end
outfile = fopen("ga.out","at");
fprintf(outfile,"\n  SCORE: %d\nHighest ever: %d\n",y7,highest_ever);
num_surviving = 0;
for (loop=1;loop<=max_pop;loop++)
{
   if (old_pool[loop].score == y7)
        {  fprintf(outfile,"\n");
           for (zebra=1;zebra<=max_chrom;zebra++)
              { fprintf(outfile,"  %s\n",word_list[old_pool[loop].chrom[zebra]]);
              }
           num_surviving++;
/*
printf("Pop member: %d is surviving in position %d\n",loop,num_surviving);
*/
   new_pool[num_surviving].score = old_pool[loop].score;
   new_pool[num_surviving].used  = old_pool[loop].used;
   for (loop2=0;loop2<=max_chrom;loop2++)
     { new_pool[num_surviving].chrom[loop2] = old_pool[loop].chrom[loop2];}
        }

}
fprintf(outfile,"Number of chromosomes carried over as is = %d\n",num_surviving);

fflush(outfile);
fclose(outfile);
 if (y7 > highest_ever) begin highest_ever = y7; end
 BLABBER(y7,save_pos,total_scores,highest_ever,worst_ever, original_high);
/* save highest one
 outfile= fopen("ga.out","at");
 for (loop=1;loop<=max_chrom;loop++) {
    fprintf(outfile,"X   %s\n",word_list[old_pool[save_pos].chrom[y]]);
    }
    fflush(outfile);
    fclose(outfile);
*/

 /* don't bother with last go 'round */
 if (GEN_NUM < GENS)
 begin
  /* force the best to survive */
 /* printf("\n forced survivor is: %d \n",save_pos); */
  printf("============================\n");
  printf(" picking new generation\n");
  printf("============================\n");
  /* PRINTPOOL(old_pool,word_list);*/
  for (x=num_surviving+1;x<=max_pop;x++)
  begin
    y = random() % total_scores +1;
    LOCATE(x,y,old_pool,new_pool);
  end
  COPY_POOL(old_pool,new_pool);
/*    CROSSOVER(old_pool, new_pool);  */
  CROSS2(old_pool,new_pool);
  COPY_POOL(old_pool,new_pool);
if (MUTATE_ON == 1)
begin

if ((GEN_NUM % MUTATE_AT)==0)
   {outfile = fopen("ga.out","at");
    fprintf(outfile,"*** Mutation occurred \n");
    fflush(outfile);
    fclose(outfile);
    MUTATE(old_pool,word_list);
   }
end
end
GEN_NUM++;
end /* while */
FINISH_UP(old_pool,y7,worst_ever,highest_ever,original_high);
printf("Last total scores was : %d\n",total_scores);
printf("Program ends\n");

end
```

BIBLIOGRAPHY

[Ackley (1987)] Ackley, D. H. (1987). *A Connectionist Machine for Genetic Hillclimbing*. Kluwer Academic Publishers.

[Baker (1987)] Baker, James Edward (1987). "Reducing Bias and Inefficiency in the Selection Algorithm". *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms*, John J. Grefenstette, ed, Cambridge, MA, 1987, pp. 14-21.

[Baker (1985)] Baker, James Edward (1985). "Adaptive Selection Methods for Genetic Algorithms". *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*, John J. Grefenstette, ed. Lawrence Erlbaum Associates, Publishers, Hillsdale, NJ, 1985, pp. 101 - 111.

[Berghel (1989)] Berghel, H. and C. Yi (1989). "Crossword Compiler-Compilation". *Computer Journal,* Vol. 32, Number 3, pp. 276-280.

[Booker (1987)] Booker, Lashon (1987). "Improving Search in Genetic Algorithms". *Genetic Algorithms and Simulated Annealing*, Lawrence Davis, ed. Morgan Kauffman Publishers, Inc., Los Altos, CA, 1987, pp. 61 - 73.

[Bridges (1987)] Bridges, Clayton L., and David E. Goldberg (1987). "An Analysis of Reproduction and Crossover in a Binary-Coded Genetic Algorithm". *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms*, John J. Grefenstette, ed, Cambridge, MA, 1987, pp. 9-13.

[Davis (1989)] Davis, Lawrence (1989). "Adapting Operator Probabilities in Genetic Algorithms", *Proceedings of the Third International Conference on Genetic Algorithms*, Morgan Kaufmann Publishers, Inc, 1989.

[Davis (1987)] Davis, Lawrence, Martha Steenstrup (1987). "Genetic Algorithms and Simulated Annealing: An Overview". *Genetic Algorithms and Simulated Annealing*, Morgan Kaufmann Publishers, Inc., Los Altos, 1987, pp. 2 - 11.

[Davis (1985)] Davis, Lawrence (1985). "Applying Adaptive Algorithms to Epistatic Domains". *Proceedings of the Ninth International Conference on Artificial Intelligence*, Volume 1, Los Angeles, CA, 1985, pp. 162-164.

[De Jong (1989)] De Jong, Kenneth A., W. M. Spears (1989). "Using Genetic Algorithms to Solve NP-Complete Problems", *Proceedings of the Third International Conference on Genetic Algorithms*, Morgan Kaufmann Publishers, Inc, 1989, pp. 124-132.

[De Jong (1988)] De Jong, Kenneth (1988). "Learning with Genetic Algorithms: An Overview". *Machine Learning*, Vol. 3, Kluwer Academic Publishers, Hingham, 1088, pp. 121 - 138.

[De Jong (1985] De Jong, Kenneth (1985). "Genetic Algorithms: A 10 Year Perspective". *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*, John J. Grefenstette, ed., Lawrence Erlbaum Associates, Publishers, Hillsdale, 1985, pp. 169 - 177.

[Eshelman (1991)] Eshelman, Larry J. and J. David Schaffer (1991). "Preventing Premature Convergence in Genetic Algorithms by Preventing Incest". *Proceedings of the Fourth International Conference on Genetic Algorithms*, Morgan Kaufmann Publishers, Inc, 1991, pp. 115 -122.

[Eshelman (1989)] Eshelman, Larry J., R. A. Caruana, and J. D. Schaffer (1989). "Biases in the Crossover Landscape", *Proceedings of the Third International Conference on Genetic Algorithms*, Morgan Kaufmann Publishers, Inc, 1989.

[Forster (1992)] Forster, J.J.H., G. H. Harris, and P.D. Smith (1992). "The Crozzle-A Problem for Automation". *Proceedings of the Symposium on Applied Computing*, ACM, NY, NY, 1992, pp. 110-115.

[Frey (1986)] Frey, Peter W. (1986). "Algorithmic Strategies for Improving the Performance of Game-Playing Programs". *Evolution, Games and Learning: Models for Adaptation in Machines and Nature*, North-Holland, NY, NY, 1986, pp. 355 - 365.

[Garey (1978)] Garey, M., D. Johnson (1979). *Computers and Intractibility: A Guide to the Theory of NP-Completeness*. Freeman, NY, 1979.

[Ginsberg (1990)] Ginsberg, Matthew L., M. Frank, M. P. Halpin, M. C. Torrance (1990). "Search Lessons Learned from Crossword Puzzles". *Proceedings: Eighth National Conference on Artificial Intelligence*, MIT Press, Cambridge, 1990pp. 210 - 215.

[Goldberg (1991)] Goldberg, David (1991). "Genetic Algorithms as a Computational Theory of Conceptual Design". *Applications of Artificial Intelligence in Engineering VI*, G. Rzevski, R. A. Adey, eds., Elsevier Science Publishing Company, NY, 1991, pp. 3 - 16.

[Goldberg (1990)] Goldberg, David E., K. Deb, and B. Korb (1990). "Messy Genetic Algorithms Revisited: Studies in Mixed Size and Scale". *Complex Systems*, Vol. 4, 1990, pp. 415-444.

[Goldberg (1989a)] Goldberg, David E., B. Korb, and K. Deb (1989). "Messy Genetic Algorithms:Motivation, Anaysis, and First Results". *Complex Systems*, Vol. 3, 1989, pp. 493 - 530.

[Goldberg (1989b)] Goldberg, David. E (1989). "Sizing Populations for Serial and Parallel Genetic Algorithms", *Proceedings of the Third International Conference on Genetic Algorithms*, Morgan Kaufmann Publishers, Inc, 1989, pp. 70-79.

[Goldberg (1989c)] Goldberg, David E. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, Reading, MA, 1989.

[Goldberg (1987)] Goldberg, David (1987). "Simple Genetic Algorithms and the Minimal Deceptive Problem". *Genetic Algorithms and Simulated Annealing*, Lawrence Davis, ed. Morgan Kauffman Publishers, Inc., Los Altos, CA, 1987, pp. 74 - 88.

[Goldberg (1986)] Goldberg, David. E. (1986). "The Genetic Algorithms Approach: Why, How, and What Next?". *Adaptive and Learning Systems: Theory and Applications*, Kumpati S. Narendra, ed., Plenum Press, NY, 1986, pp. 247 - 253.

[Grefenstette (1988)] Grefenstette, John J. (1988). "Credit Assignment in Genetic Learning Systems". *Proceedings of AAAI-88, Seventh National Conference on Artificial Intelligence*, 1988, pp. 596 - 600.

[Grefenstette (1987)] Grefenstette, John J. (1987). "Incorporating Problem Specific Knowledge into Genetic Algorithms". *Genetic Algorithms and Simulated Annealing*, Lawrence Davis, ed. Morgan Kauffman Publishers, Inc., Los Altos, CA, 1987, pp. 42 - 60.

[Grefenstette (1985b)] Grefenstette, John. J. (1986). "Optimization of Control Parameters for Genetic Algorithms". *IEEE Transactions on Systems, Man and Cybernetics*, IEEE Press, 1986, Vol 16 No. 1, pp. 122 - 128.

[Grefenstette (1985a)] Grefenstette, John J., and J. M. Fitzpatrick (1985). "Genetic Search with Approximate Function Evaluations". *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*, John J. Grefenstette, ed. Lawrence Erlbaum Associates, Publishers, Hillsdale, NJ, 1985, pp. 112-120.

[Harris (1993b)] Harris, Geoff, John Forster, Richard Rankin (1993). "Basic Blocks in Unconstrained Crossword Puzzles". To appear, *Proceedings of the 1993 Symposium on Applied Computing*, ACM Press, NY, 1993.

[Harris (1993a)] Harris, G. H., J.J.H. Forster, P.D. Smith (1993). "The Crozzle- A Lexicographic NP-Complete Problem", in preparation, 1993.

[Harris (1992d)] Harris, G. H. and J.J.H. Forster (1992). "On the Solution S(k,n) to a Class of Crossword Puzzles". *The Computer Journal*, 35, pp. A177-A180.

[Harris (1992c)] Harris, G. H., J. Spring and J.J.H. Forster (1992). "An Efficient Algorithm for Puzzle Solutions". *The Computer Journal*, 35, pp. A181-A183.

[Harris (1992b)] Harris, G. H. (1992) private communication.

[Harris (1992a)] Harris, G., D. Roach, H. Berghel, and P.D. Smith (1992). "Dynamic Crossword Slot Table Implementation". *Proceedings of the 1992 Symposium on Applied Computing*, ACM, NY, NY, 1992, pp. 95-98.

[Harris 1990b] Harris, Geoff (1990). "Generation of Solution Sets for Unconstrained Crossword Puzzles". *Proceedings of the 1990 Symposium on Applied Computing*, IEEE Press, Los Alamitos, CA, 1990, pp. 214 -219.

[Harris 1990a] Harris, G. H., and J.J.H. Forster (1990). "On the Bayesian Estimation and Computation of the Number of Solutions to Crossword Puzzles". *Proceedings of the 1900 Symposium on Applied Computing*, IEEE Press, Los Alamitos, CA, 1990, pp. 220 -222.

[Hill (1992)] Hill, A. and C. J. Taylor (1992). "Model-Based Image Interpretation Using Genetic Algorithms". *Image and Vision Computing,* Vol. 10, No. 5, Butterworths, pp. 295 -300.

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

[Holland (1973)] Holland, John H. (1973). "Genetic Algorithms and the Optimal Allocation of Trials". *SIAM Journal of Computing*, Vol. 2, No. 2, 1973, pp. 88 - 105.

[Holland (1971)] Holland, John H. (1971). "Processing and Processors for Schemata". *Associative Information Techniques*, Edwin L. Jacks, ed. American Elsevier Publishing Company, Inc., NY, NY, 1971, pp. 127 - 146.

[Huntley (1991)] Huntley, Christopher L., D. E. Brown 1991. "A Parallel Heuristic for Quadratic Assignment Problems". *Computers and Operations Research.* Pergamon Press, NY, 1991, pp. 275 - 289.

[Jog (1989)] Jog, Prasanna, J. Y. Suh, D. Van Gucht (1989). "The Effects of Population Size, Heuristic Crossover and Local Improvement on a Genetic Algorithm for the Travelling Salesman Problem". *Proceedings of the Third International Conference on Genetic Algorithms*, Morgan Kaufmann Publishers, Inc, 1989, pp. 110-115.

[Liepins (1987)] Liepins, G. E., and M. R. Hilliard (1987). "Greedy Genetics". *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms*, John J. Grefenstette, ed, Cambridge, MA, 1987, pp. 90-99.

[Mauldin (1984)] Mauldin, Michael L. (1984). "Maintaining Diversity in Genetic Search". *Proceedings of the National Conference on Artificial Intelligence*, AAAI, Austin, TX, 1984, pp. 247-250.

[Michaelewicz (1992)] Michalewicz, Zbigniew. *Genetic Algorithms + Data Structures = Evolution Programs*, Springer-Verlag, New York, 1992.

[Michalski (1983)] Michalski, R., ed. (1983). *Machine Learning: An Artificial Intelligence Approach*, Tioga Press, Palo Alto, 1983.

[Mazlack (1976)] Mazlack, L.J. (1976). "Machine Selction of Elements in Crossowrd Puzzles - An Application in Computational Linguistics". *SIAM Journal of Computing*, Vol. 5, No. 2, pp. 51-72.

[Papadimitrious (1977)] Papadimitrious, C. H., and K. Steiglitz (1977). "On the Complexity of Local Search for the Travelling Salesman Problem". *SIAM Journal of Computing*, Vol. 6, 1977, pp. 78 - 83.

[Pham (1991)] Pham, D. T. and H. H. Onder (1991). "An Expert System for Ergonomic Design Using a Genetic Algorithm". *Applications of Artificial Intelligence in Engineering VI*, G. Rzeveski, and R. A. Adey, eds., Elsevier Science Publishing Company Inc, NY, pp. 288-297.

[Rankin (1993b)] Rankin, Richard, R. Wilkerson, G. Harris, L.J. Spring, (1993). "A Hybrid Genetic Algorithm for an NP-Complete Problem With an Expensive Evaluation Function". To appear, *Proceedings of the 1993 Symposium on Applied Computing*, ACM Press, NY, 1993.

[Rankin (1993a)] Rankin, Richard, G. Harris, L.J. Spring (1993). "A Non-Standard Hybrid Genetic Algorithm", in preparation, 1993.

[Reynolds (1991)] Reynolds, Robert G. (1991). "Version Space Controlled Genetic Algorithm". *Proceedings: Second Annual Conference on AI, Simulation and Planning in High Autonomy Systems*, IEEE Computer Society Press, Los Alamitos, 1991, pp. 6 - 14.

[Reynolds (1990)] Reynolds, Robert G. (1990). "The Control of Genetic Algorithms Using Version Spaces". *Proceedings of the Second International Conference on Tools for Artificial Intelligence*, IEEE Computer Society Press, Los Alamitos, 1990, pp. 342 - 348.

[Rizki (1991)] Rizki, Mateen M., L. A. Tamburino, M. A. Zmuda (1991). "Applications of Learning Strategies to Pattern Recognition". *SPIE Vol. 1469: Applications of Artificial Neural Networks II*, 1991, pp. 384 - 391.

[Sambridge (1992)] Sambridge, Malcolm and Guy Drijkoningen (1992). "Genetic Algorithms in Seismic Waveform Inversion". *Geophysics Journal International*, Oxford Press, pp. 323-342.

[Sano (1992)] Sano, Chiharu (1992). "Hybrid of (ID3 extension + Backpropogation) Hybrid & (Case-Based Reasoner + Grossberg Net) Hybrid with Economics Modelling Controlled by Genetic Algorithm". *SPIE Vol. 1707: Applications of Artificial Intelligence X: Knowledge-Based Systems*, 1992, pp. 180 - 194.

[Schaffer (1989)] Schaffer, J. David, R. Caruana, L. J. Eshelman, and R. Das (1989). "A Study of Control Parameters Affecting Online Performance of Genetic Algorithms for Function Optimization", *Proceedings of the Third International Conference on Genetic Algorithms*, Morgan Kaufmann Publishers, Inc, 1989.

[Schaffer (1987)] Schaffer, J. David (1987). "An Adaptive Crossover Distribution Mechanism for Genetic Algorithms". *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms*, John J. Grefenstette, ed, Cambridge, MA, 1987, pp. 36-40.

[Schaffer (1985)] Schaffer, J. David (1985). "Multiple Objectives with Vector Evaluated Genetic Algorithms". *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*, John J. Grefenstette, ed. Lawrence Erlbaum Associates, Publishers, Hillsdale, NJ, 1985, pp. 93 - 100.

[Smith (1983)] Smith, P. D. (1983). "XENO: Computer-Assisted Compilation of Crossword Puzzles". *The Computer Journal*, Vol. 26, No. 4, pp-296-302.

[Smith (1981)] Smith, P. D., and Steen, S.Y. (1981). "Prototype Crossword Compiler". *The Computer Journal*, Vol. 24, No. 2, pp. 107-111.

[Spears (1992)] Spears, William M, (1992). "Crossover or Mutation?". unpublished manuscript.

[Spring (1993)] Spring, Jo (1993). "Benchmarking Automated Solution Generators for the Crozzle". To appear, *Proceedings of the 1993 Symposium on Applied Computing*, ACM Press, NY, 1993.

[Syswerda (1989)] Syswerda, Gilbert (1989). "Uniform Crossover in Genetic Algorithms", *Proceedings of the Third International Conference on Genetic Algorithms*, Morgan Kaufmann Publishers, Inc, 1989.

[Suh (1987)] Suh, Jung Y., and Dirk Van Gucht (1987). "Incorporating Heuristic Information into Genetic Search". *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms*, John J. Grefenstette, ed, Cambridge, MA, 1987, pp. 100 - 107.

[Szarkowicz (1991)] Szarkowicz, Donald S. (1991). "A Multi-Stage Adaptive-Coding Genetic Algortihm for Design Applications". Proceedings of the 1991 Summer Computer Simualtion Conference. Baltimore, MD, pp. 138-144.

[Tamburino (1992)] Tamburino, Louis A., M. M. Rizki (1992). "Performance-Driven Autonomous Design of Pattern-Recognition Systems". *Applied Artificial Intelligence*, Vol. 6. Hemisphere Publishing Company, Washington, D.C., 1992, pp. 59 - 77.

[Tamburino (1990)] Tamburino, Louis A., M. M. Rizki (1990). "Applications of Hybrid Learning to Automated System Design". ***Proceedings: AI, Simulation, and Planning in High Autonomy Systems***, Bernard Zeigler, J. Rozenblit, eds., IEEE Computer Society Press, Los Alamitos, 1990, pp.176 - 183.

[Thangiah (1992)] Thangiah, Sam R. and Kendall E. Nygard (1992). "School Bus Routing Using Genetic Algorithms". *Proceedings of SPIE,* Bellingham, WA, SPIE, pp. 387-398.

[Young (1990)] Young, R. A., A. Reel (1990). "A Hybrid Genetic Algorithm for a Logic Problem"". *ECAI90:Proceedings of the Ninth European Conference on Artificial Intelligence*. Pitman, London, 1990, pp. 744 - 746.