

**A GENETIC ALGORITHM TO EXTRACT N-TUPLES
FROM A GIVEN SET OF WORDS**

By

RAVI B. MANDADI

Bachelor of Engineering

Osmania University

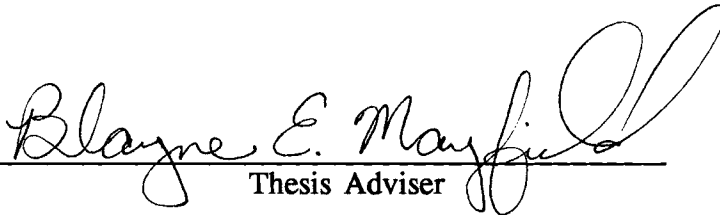
Hyderabad, India

1989

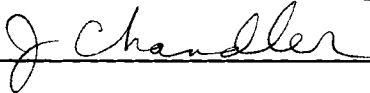
Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May, 1995

A GENETIC ALGORITHM TO EXTRACT N-TUPLES
FROM A GIVEN SET OF WORDS

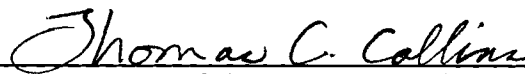
Thesis Approved:



Thesis Adviser







Dean of the Graduate College

ACKNOWLEDGMENTS

I wish to thank my major adviser, Dr. Blayne Mayfield for his guidance, support, and inspiration. I feel that I would have not made it through these last three years without his friendship and encouragement. I would also like to thank my other committee members Dr. J. P. Chandler, for his constructive suggestions and supportive guidance, and Dr. Lu for her time.

More over, I wish to thank Dr. George Sabbagh for his support, friendship, and for allowing me to use his computer and necessary software to write my thesis. My paper would not be complete without giving my gratitude to Mrs. Griffin for helping me with my grammar.

3.1.8.1.2 The Driver/Main program	42
3.1.8.2 Approach 2: Using the LibGA application tool . . .	43
3.1.8.2.1 The Main Program	43
3.1.8.2.2 The Configuration file	45
3.1.8.2.3 The User Data file	45
3.2 Description of the Combinatorial/Exhaustive Search Approach	46
3.3 General description of the Trial and Error Approach	47
3.4 About the Implementation Platform	48
4. RESULTS AND OBSERVATIONS	49
4.1 The Performance of the Genetic Algorithm	50
4.2 The Execution Time	50
4.3 The Solution Quality	56
4.4 Biased Initialization Vs. Random Initialization	59
4.5 Comparison of the two implementations of the genetic algorithm	62
4.6 Additional graphs presented in Appendix A	64
5. SUMMARY AND CONCLUSIONS	66
5.1 Conclusions	66
5.2 Future Work	67
BIBLIOGRAPHY	69
APPENDIXES	
Appendix A--Graphs	72
Appendix B--Source code of the genetic algorithm implemented in C	77
Appendix C--Source code for the genetic algorithm implemented using libGA application	96

LIST OF FIGURES

Figure	Page
1. The genetic representation of a tuple	5
2. The genetic cycle	6
3. The crossover operation	12
4. The mutation operation	13
5. Outline of the genetic algorithm	26
6. Genetic code	27
7. A sample solution space	29
8. Performance graph for words of length 6	51
9. Performance graph for words of length 8	52
10. Performance graph for words of length 10	53
11. Performance graph #1 for words of increasing length	54
12. Performance graph #2 for words of increasing length	55
13. Solution quality for words of length 6	56
14. Solution quality for words of length 10	57
15. Solution quality for words of increasing length	58
16. Comparison of performance of two initialization techniques	60
17. Solution quality corresponding to the graph of Figure 16	61
18. Convergence when not using elitism	63
19. Convergence when using elitism	64

20. Performance graph #3 for words of increasing length 72

21. Solutions corresponding to the graph in figure 20 73

22. Solutions corresponding to graph in figure 12 74

23. Comparison of performance of two initialization techniques 75

24. Solution quality corresponding to the graph of figure 23 75

25. Comparison of performance of two initialization techniques 76

26. Solution quality corresponding to the graph of figure 25 76

CHAPTER 1

1. INTRODUCTION

Living organisms are perfect problem solvers due to their versatility. These organisms come by their abilities through the apparently undirected mechanisms of evolution and natural selection. Genetic algorithms are search algorithms that are based on the mechanics of natural selection and natural genetics. By harnessing the powers of natural selection and evolution, genetic algorithms are able to *breed* solutions to problems whose structures are not fully understood. Although genetic algorithms in their present form were invented by John Holland [HOL73], the idea of simulating biological evolution for optimization dates back to the 1960s. Genetic algorithms are optimized by maximizing a fitness function or minimizing a penalty function.

Genetic algorithms make it possible to explore a far greater range of potential solutions to a problem than do conventional programs through the use of two primary processes: natural selection and reproduction. Genetic algorithms model the way in which biological genetic processes seem to operate. This is accomplished by using three processes known as genetic operators. The definition of the genetic operators in the context of genetic algorithms was obtained by studying the natural evolution of biological beings or organisms. Evolution is a powerful process that created the form of life as we know today. A few billion years ago life existed in the form of extremely primitive organisms that contained very little genetic information. These biological organisms

exhibited only primitive functions that were enough to sustain them. From generation to generation, these organisms became more complex and better able to adapt themselves to the changing environment. The algorithm that achieves this function uses three natural biological processes: reproduction, mutation, and mating. Following is a description of the roles of these three biological processes in natural evolution and how these processes are adapted to solve problems in the field of genetic algorithms.

The biological environment in genetic algorithms is simulated by use of a fitness function. The environment in biological evolution provides the standards which organisms must meet in order to survive. As stated by Goldberg [GOL89], "In natural populations, fitness is determined by a creature's ability to survive predators, pestilence, and other obstacles to adulthood and subsequent reproduction." Fitness plays a similar role in genetic algorithms. Just as organisms in a biological environment evolve to improve their ability to survive, *chromosomes* in genetic algorithms evolve to attain higher fitness values and become more adept at solving the problem. The fitness of a chromosome is a measure of its ability to solve the problem. Chromosomes in the context of genetic algorithms consist of a group of string values, each of which represents a characteristic called a *gene*. The chromosome represents a possible solution to the problem. The fitness function is the criterion for choosing the best chromosomes to use in creating a new generation. This selection strategy is based on the Darwinian principle of the survival of the fittest. Just as surviving organisms reproduce in nature, the selected chromosomes will be mated to reproduce. The *mating* or *crossover* process is accomplished by combining different parts of different chromosomes to produce *offspring*

chromosomes with characteristics derived from their parents. Occasionally one of the string values of a randomly selected chromosome is altered. This process simulates a process which is the biological equivalent of a *mutation*. This evolution process, in short, consists of repeatedly applying the three genetic operators, mating, mutation, and selection, until a good solution is obtained, i.e, in biological terms, when the chromosomes evolve sufficiently to contain enough of the desired features or characteristics. This condition for termination signifies the convergence to a solution, since at this point the population will be dominated by superior chromosomes.

Genetic algorithms are used in optimization problems, in which a *fitness function* facilitates the selection of the best chromosomes in a population. The solution to an optimization problem is to maximize an objective/fitness function $J(x)$ on a set X , where X is the entire solution space. *Fitness* came from an analogy with Darwinism that underlies this whole approach of simulating an evolution to guarantee the survival of the fittest. The design and implementation of the objective function is problem dependent and is described in Chapter 3. The process of solving this optimization problem using evolution is explained in the following five steps. (1) First the population size is fixed at some value n . Then n elements x_i are chosen at random from X to form the individuals of the first generation. (2) For each of the individuals x_i , the value $J(x_i)$ is computed. (3) Then for each of the individuals x_i , the survival probability p_i is computed as the ratio of the individual's fitness to the sum total of the fitnesses of all the individuals. A random number generator generates each individual x_i with the probability p_i and also selects individuals to populate the next generation. This process is explained

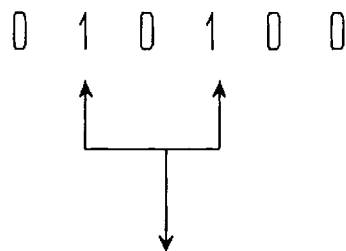
in Chapter 3. The formula for p_i actually means that survival probability is proportional to fitness, $F(x_i)$. (4) At this point, the n individuals that will populate the next generation are determined. Some of these individuals remain the same as in the previous generation and others are computed. A new individual is obtained by running the random individual generator twice to get two individuals x_j and x_k and then combining these two individuals according to some combination rule. The combination rule involves the use of genetic operators whose operation and implementation are described in Chapter 2. This process is repeated until an entirely new generation of size n is obtained. (5) Steps 2 through 4 are repeated with the individuals of the new generation.

It is desirable that each generation contain individuals that solve the problem better than their predecessors. In other words, each population set will contain individuals whose *fitness* is greater than that of their predecessors. Thus each generation produces successively better solutions to the problem. According to the schema theorem explained by Michalewicz [MIC92], progressive improvement of the solutions is guaranteed when binary string representations of the solution are used. The key to a successful genetic algorithm is to determine appropriate fitness criteria.

Genetic algorithms have been applied successfully to a wide range of real-world problems such as, scheduling, machine learning, partitioning of graphs, the traveling salesman problem, and the transportation problem to name a few. The genetic codes used in solving each of these problems were different, since the representation issues are problem specific. There are no rules for designing a genetic code.

The extraction of n -tuples plays an important role in designing a perfect hashing

scheme. Various hashing schemes that use n-tuples to build perfect hash functions are described in Chapter 2. The problem of extracting these n-tuples from a given word list, using a genetic algorithm is addressed in this paper and is stated as follows. From a given list of words, it is required to extract a list of n-tuples. Each n-tuple is an ordered list of n characters picked from each word of the list. Characters must be picked from distinct positions in a word and the same positions used for all the words in the list. Furthermore, the n-tuples must be unique and the number of characters picked to form the tuples must be minimal. Since the n-tuples are potential solutions to the problem, a genetic representation of an n-tuple must be defined. The genetic representation of the n-tuple is defined by a binary string 's', of length equal to the word length of the shortest word, such that if $s[i] = 1$ then characters from position i are selected to form the tuple. The *size* of the tuple is the number of 1's in the string 's'. For example, consider the words, CONST, LABEL, VALKE, PACKED. The genetic code, 010100, would represent the list of 2-tuples, (O,S), (A,E), (A,K), (A,K), obtained by selecting characters at positions 2 and 4 from the word list. The genetic code (chromosome) for this selection is pictured in Figure 1.



The 1's indicate the selection of characters
in positions 2 and 4

Figure 1: The genetic representation of a tuple

The fitness function is designed to rate these tuples according to their size and *overlap*. The *overlap* in this context is the total number of duplicate entries among the list of tuples. For example the overlap of the list of tuples mentioned above is 2 since there are two duplicate entries, namely (A,K) and (A,K).

The initial generation will contain a population of a random mix of these chromosomes. The three genetic operators, selection, mating and mutation, are applied randomly to this population to obtain the new population. These genetic operators are described in Chapter 2. This process is repeated to create each new generation until a terminating condition is reached, such as a maximum number of generations, or until the individuals of the new generation reach a desired level of fitness. The *fitness* in this context refers to the ability of a member of the generation to solve the problem. While it may seem that this search for a solution is random, in fact, the improvement of fitness values in each generation indicates that the genetic algorithm provides an effective directed search technique. The basic genetic cycle that creates a new generation by applying the three genetic operators on an existing population is shown in Figure 2.

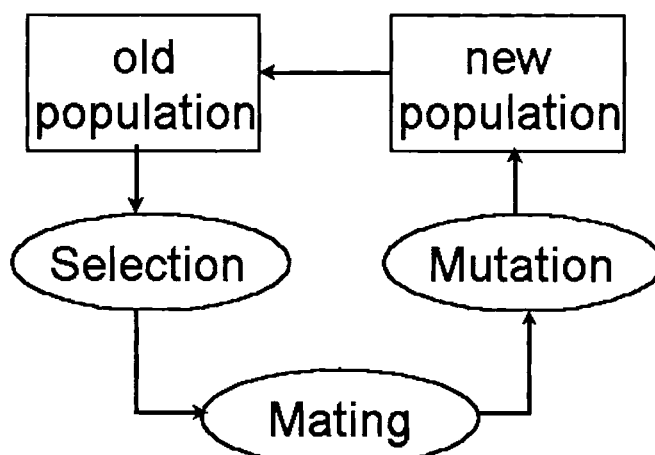


Figure 2: The genetic cycle

The goal of the fitness function is to select not only the tuples with least amount of overlap but also of the smallest size. As in this case, many practical problems require that more than one constraint be satisfied. Constraints are usually classified as equality or inequality relations. The goal of the fitness function is to minimize the overlap of the tuples; subject to the constraint that among tuples of the same overlap, smallest tuples are selected. The overlap as well as the size must be minimal for a good solution, but in case of a tie, the overlap is used for rating the solution. The implementation of constraints (explained in Chapter 3) on solutions play an important role in designing a chromosome representation of solutions to the problem.

Decoders are used by fitness functions to identify chromosomes that violate one or more constraints, during the selection process. The decoder identifies and isolates the individuals that violate constraints and repairs or destroys them. Constraints that cannot be violated can be implemented by imposing great penalties in the fitness function on individuals that violate them, by imposing moderate penalties, or by creating *decoders* of the representation that avoid creating individuals that violate the constraint. Each of these methods has its advantages and disadvantages. If a high penalty is incorporated into the evaluation routine and the domain is one in which production of an individual violating the constraint is likely, then there exists a risk of creating a genetic algorithm that spends most of its time evaluating illegal individuals. Furthermore, there is a risk of premature convergence to a solution that is not optimal, since the likely paths to better individuals require the production of illegal individuals as intermediate structures, and the penalties for violating the constraint make it unlikely that such intermediate structures

will reproduce. On the other hand, imposing moderate penalties may result in producing individuals that violate the constraint but are rated better than those that do not because the rest of the evaluation function can be satisfied better by accepting the moderate constraint penalty than by avoiding it. A decoder built into the evaluation procedure can be designed to intelligently avoid building an illegal individual but running it is frequently computation-intensive. In the implementation of the constraints discussed in Chapter 3, a repair procedure is designed and incorporated in the evaluation scheme. This procedure admits illegal individuals but then immediately identifies them and sets a severe penalty on such individuals to avoid violation of the constraint.

Chapter 2 presents a review of related work and the history of genetic algorithms. It also defines the genetic operators and terms that are commonly used. The design and implementation of the combinatorial approach to solving the problem is explained in Chapter 3. In Chapter 4, the performance of this approach is compared with the performance of the genetic approach and observations are made. The *execution time* to arrive at the solution is computed for both methods and graphed for various randomly generated data sets. The quality of the output in terms of tuple size and overlap size is evaluated and compared for the solutions obtained by both these methods.

CHAPTER 2

2. LITERATURE REVIEW

2.1 History

Biological analogies have been part of the science and the lore of computation since the 1940s. The concept of evolution in molecular biology has been extended to include computing. Landauer [LAD71] made it clear that there are parallels between evolutionary and computer processes.

According to Lander [LAN91], the field of biology is rapidly becoming much more computational and analytical. Molecular biologists determined that the gene is made up of DNA, deoxyribonucleic acid. Lander states that the DNA double helix, to computer scientists, is a clever, robust, information storage and transmission system. Damian [DAM91] describes the structure of a DNA double helix as a polymer consisting of four elements, A, T, C, and G (adenine, thymine, cytosine, and guanine). According to him, the four letter alphabet of DNA can encode messages of arbitrary complexity. Fast matching algorithms have been applied in sequence analysis [PAV92] of DNA sequences consisting of the four elements mentioned above.

Analogies between computing and biology are not a mere coincidence. Both biological genes and computers record, copy, and disseminate information. Douglas Hofstadter of Indiana University showed this clearly [HOF85] by demonstrating that the action of DNA and RNA during the reproduction of the living cell can be interpreted as

an example of a self-reproducing Turing machine. Lawrence Hunter [HUN91] says that the early AI systems functioned in the field of molecular biology and were later adapted and modified to apply in different fields of computing such as, computational linguistics, that can be applied to text and DNA sequences, neural networks, qualitative modelling, hierarchical pattern recognition, case-based reasoning, visual processing, expert systems, knowledge-based systems, minimal length encoding, etc.

The beginnings of genetic algorithms can be traced back to the early 1950s. Goldberg [GOL85] says that during this time several biologists used computers for simulations of biological systems. The work done in the late 1960s and early 1970s at the University of Michigan under the direction of John Holland led to genetic algorithms as they are known today. Holland was inspired by a Darwinian notion of evolution in which only the fittest survive. He proposed that a learning machine's search for a good learning strategy be organized as the breeding of many strategies in a population of candidates, rather than as the construction and refinement of a single strategy. Holland and his students called their searches *reproductive plans* which later became popularly known as genetic algorithms, after Holland's seminal book published in 1975 [HOL75]. In 1989 David Goldberg of the University of Alabama published "Genetic Algorithms" [GOL89] a book, that demonstrated a solid scientific basis for the field and cited more than 73 successful applications.

2.2 Terms and notations

The term *chromosome* is analogous to a biological organism. Just as the process

of natural evolution works on a population consisting of biological organisms, so the computer model of evolution works on a population of chromosomes, which in the context of computers, denotes a string of characteristics. A gene is the basic building component of a chromosome, which in the context of computing is a single element of the string of characteristics representing the chromosome.

Just as surviving organisms reproduce in nature, so do selected chromosomes in a genetic pool. The selection of the chromosomes that are mated to produce offspring is based on their high fitness values. This process of selecting the chromosomes that yield better fitness when input to a fitness function is called *natural selection* or biased reproduction. *Fitness function*, also known as *evaluation function*, plays the role of the environment, rating potential solutions in terms of their fitness. Consider for example the problem of optimizing a simple function of one variable, $f(x) = x^2 - 2x + 1$, in the domain $[1, 10]$. If the evaluation function is chosen to be equivalent to the function f , then the 3 chromosomes, $v_1 = (0101_2)$, $v_2 = (1010_2)$, $v_3 = (0110_2)$ would be rated as follows,

$$\text{Fitness}(v_1) = f(v_1) = f(0101_2) = f(5) = 16$$

$$\text{Fitness}(v_2) = f(v_2) = f(1010_2) = f(10) = 81$$

$$\text{Fitness}(v_3) = f(v_3) = f(0110_2) = f(6) = 25$$

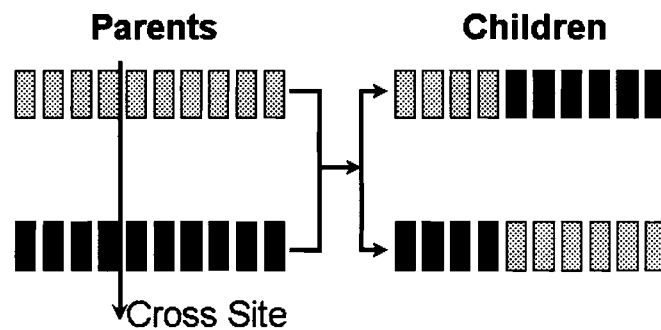
The process of mating involves combining different parts of different chromosomes to produce new chromosomes, also known as *crossover*. This process is explained in the next section.

Mutation is the alteration of the value of an arbitrary gene of a chromosome.

The concept of evolution is simulated on a computer by random applications of these three operators: selection, crossover, and mutation.

2.3 The Role Of Mating/Crossover

The process of mating ensures the mixing and recombination among the genes of their offspring. Thus this process is responsible for transferring the combined properties



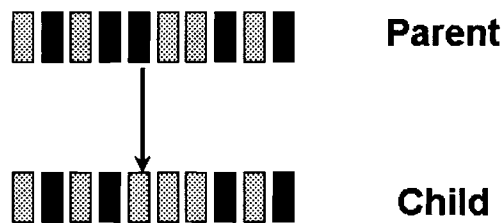
Crossover/Mating: Crossover never changes the value of genes. It simply rearranges existing gene values in different ways

Figure 3: The crossover operation

of the mating genes of the current generation to their offspring in the next generation. The conventional scheme of mating is the swapping of characteristics between two chromosomes across a random point within the length of both chromosomes. Many variations of this scheme are permitted as long as a monotonic increase in the quality of genes in succeeding generations is ensured on an average over all the chromosomes in a population. The process is demonstrated in Figure 3.

2.4 The Role Of Mutation

Mutation in the genetic algorithm is an arbitrary change in a given characteristic in the chromosome. This process is demonstrated in Figure 4. The mutation rate is set at a very low level as its main function is to restart a process of evolution that has stalled. Mutation also helps to maintain pattern diversity and introduce new characteristics into the population. Pattern diversity in the population corresponds to the



Mutation: Mutation changes the gene value of an arbitrarily selected gene of a chromosome.

Figure 4: The mutation operation

breadth of the searched domain. Loss of this diversity results in an undesirable, premature convergence of the algorithm to a non-optimal solution.

2.5 Natural Selection (Biased reproduction)

The process of natural selection determines which members of a population survive to reproduce and is done by using a biased, random-selection methodology.

Parents are randomly selected from the current population in such a way that the *fittest* genes in the population have the greatest chance of being selected. Holland [HOL75] suggested a scheme for this type of selection which involved a linear search through a roulette wheel with slots weighted in proportion to string fitness values. The roulette wheel implements the natural selection process in a way that incorporates the Darwinian notion of the survival of the fittest. Using natural selection of the *fittest* points in a solution space forces the algorithm to move in the most promising direction in its overall search.

2.6 Fitness Concept

The genetic algorithm exploits the higher-payoff, or *target*, regions of the solution space due to the fact that successive generations of reproduction and crossover produce increasing numbers of strings in those regions. A region is a schema which is defined as follows. Let Ω be the set of all strings of some fixed length over some alphabet Σ , then $|\Omega| = |\Sigma|^n$. let Ψ be the power set of Ω . Then the schema H is defined by Vose [VOS91] as a function that maps into the set {true, false} according to the rule,

$$H(x) = \text{true iff } x \in M \text{ where } H \text{ is the schema describing the subset (of } \Omega) M \in \Psi.$$

For example let Ω be the set of all strings of length 2 over the alphabet $\Sigma = \{0,1\}$. Then the set $\Omega = \{00, 01, 10, 11\}$ and the power set of Ω , $\Psi = \{\phi, \{00\}, \{01\}, \{10\}, \{11\}, \{00,01\}, \{00,10\}, \{00,11\}, \{01,10\}, \{01,11\}, \{10,11\}, \{00,01,10\} \dots\dots\dots, \{00,01,10,11\}\}$. Any element of Ψ represents a schema. For instance $\{01, 11\} \in \Psi$ is

described by the schema *1 which maps binary strings of length two into the set {true, false} as follows,

$$*1(x) = \text{true iff } x \in \{01, 11\}.$$

The * in the above equation represents a *don't care* symbol which can take on any single character from the alphabet. This notation was adopted by Vose [VOS91]. As an example the schema 1*0* would contain the strings, 1000, 1001, 1100, 1101.

According to Holland [HOL92], a genetic algorithm casts a net over a landscape of potential solutions, where the rate at which the genetic algorithm samples different regions corresponds to the regions' average elevation or fitness. The algorithm favors the fittest genes as parents, and so above-average genes will have more offspring in the next generation. The remarkable ability of genetic algorithms to focus their attention on the most promising parts of a solution space is a direct outcome of their ability to combine genes containing partial solutions.

The choice of a fitness function depends on the nature of the optimization problem. Consider for example the problem of solving the two simultaneous equations $ax + by = c$ and $dx + ey = f$. The solution domain is the set of equations made up of either the addition, subtraction, or multiplication of the six constants a, b, c, d, e and f . Since the chromosome should represent any equation in the solution domain, it is denoted by a binary tree of nodes containing the coefficients. If we choose values for the coefficients, such as $a=1, b=2, c=4, d=0.5, e=2$ and $f=3$, then the solution is $x = 2$. So let the fitness of a chromosome be defined as the sum of the squares of the differences between the correct answer 2 and the selected chromosome. The best

chromosome will have a fitness of 0 and the worst will be a large number. The best chromosome will represent the solution for all values of a through f with a high probability.

Game theory suggests that each player should minimize the maximum damage the other player can inflict. For example, in the simple game known as prisoner's dilemma, Michalewicz [MIC92] describes two prisoners that are held in separate cells and are unable to communicate with each other. Each prisoner is asked to defect and to betray the other prisoner. If both defect, they are both tortured. If one defects, he is rewarded and the other prisoner is punished. Thus, if any one prisoner selfishly chooses defection, he is guaranteed a higher payoff than if he chooses cooperation. But if both defect at the same time then both are worse off than if they had cooperated. The prisoner's dilemma is to decide whether to defect or cooperate with the other prisoner. Applying the genetic algorithm to such problems requires translating possible strategies into strings. One simple way is to base the next response on the outcome of the last three plays. The value of each gene would be either 1 or 0 depending on whether the preferred response to its corresponding history was cooperation or defection. The fitness of each string could be heuristically constructed as the average of the payoffs its strategy received after repeated play.

For problems in which various factors affect the optimization of the solution, the fitness function might be constructed as the weighted sum of all the factors, the weights being dependent on the nature of the problem.

Thus there are no general rules for the design of the fitness function. The fitness

function for an optimization problem is usually designed at a heuristic and intuitive level, taking into consideration the factors that play a role in determining an optimal solution to the problem.

2.7 Need for Extraction of n-tuples

In this paper, an n-tuple is an ordered list obtained by extracting characters at n different fixed positions in a word. This implies that the word has to be at least n characters long. Let $w = c_1c_2c_3\dots c_p$ be a word containing characters c_1 through c_p , where $p \geq n$. An n-tuple extracted from this word is defined as, $t_n = (c_i, c_j, c_k, \dots)$ where $1 \leq i < j < k < \dots \leq p$ and $|t_n| = n$. For an arbitrary list of words, an n-tuple is extracted from each word with the same values of i, j, k,.. to form a set of n-tuples. The set of n-tuples extracted from an arbitrary list of words is said to be unique when no two tuples in the set are identical. Chang [CHI91] states that finding a good heuristic algorithm to extract a unique n-tuple for an arbitrary list of words with the least amount of required time still remains an open problem. In his paper Chang [CHI91] says, "Up to now researchers have proposed many perfect hashing schemes using extracted n-tuples. They all used trial and error to find the needed n-tuples." In this paper Chang introduces a letter-oriented perfect hashing scheme with a space complexity of :

$$O\left(\frac{4\omega \lceil n/2 \rceil}{N}\right)$$

,where ω is the cardinality of the set of characters appearing in all extracted n-tuples, N denotes the number of words hashed and n is the tuple length. The time spent in finding a perfect hashing function is based on the time complexity of the Matrix Compression algorithm used, which is :

$$O\left(\frac{N^2}{\rho^2}\right)$$

where ρ the compression rate, $1 < \rho \leq 1$.

From this equation it is apparent that the space needed by this hashing scheme depends on the size of the extracted tuples n . The genetic algorithm used to extract these tuples attempts to optimize this parameter n . Numerous hashing schemes can be cited where extraction of n-tuples from an arbitrary list of words played a key role. The only method suggested for the extraction of these n-tuples involved repeated attempts at guessing a possible n-tuple on a trial and error basis. An overview of various hashing schemes that use extracted n-tuples is given below.

2.7.1 Application to Perfect Hashing Schemes

Up to now researchers have proposed many perfect hashing schemes using extracted n-tuples. Their performance relies on the low cardinality of the extracted n-tuples. The cardinality of a tuple is the number of characters that it contains. The disadvantage of perfect hashing functions to date has been that the process for

constructing them is slow. Cichelli's algorithm [CIC80] searches a table that contains a calculated value for each letter in the word (key) alphabet. Cichelli's algorithm has the advantage of simplicity but it also has an exponential expected time complexity, $O(e^r)$, where r is the number of keys in the set to be stored. Values associated with the letters from the key are used in determining the storage address for the corresponding record. The minicycle algorithm described by Sager [SAG85] grew out of an attempt to optimize Cichelli's algorithm for generating perfect hash functions. The minicycle algorithm, which is currently the most computationally efficient, has a worst case time complexity of $O(r^6)$ observed by Marshall Brain [BRA89].

Marshall Brain [BRA89] presented a near perfect hashing scheme on large word sets based on a modification of Cichelli's algorithm. The improved procedure was the result of examining the original algorithm for the causes of its sluggish performance and then modifying them. Similar to the original Cichelli's algorithm, a table that contains a calculated value for each letter in the key's alphabet was maintained. Then the hashing function,

$$h(key) = \frac{table_{value-of-first-letter}(key) + table_{value-of-last-letter}(key)}{string-length(key)}$$

was used. An improvement was made in the ability to backtrack intelligently in case of a collision, which is the case when the first and last characters of two equal-length words are the same. Though this improved algorithm had the advantage of efficiency and

simplicity, the great disadvantage was the larger storage usage than other methods.

In a new approach, described by Chang [CHA91], to design a perfect hashing scheme, the assigned address of each keyword was determined as the following form,

$$\text{address} = v_1 (\text{the keyword's } i\text{th character}) + v_2 (\text{the keyword's } j\text{th character})$$

where v_1 and v_2 were two integer-valued functions defined on the set of twenty-six English letters. The letter value assignments for v_1 and v_2 were determined by a letter-oriented merging-and-exchanging algorithm. The loading factor depended heavily on the extracted pairs. When a pair of extracted letters was not sufficient to distinguish all the keys, this scheme failed.

Chang and Lee [CHA86] describe a minimal perfect hashing function suitable for letter-oriented keys, based on the Chinese Remainder Theorem. This hash function was a result of modifying the algorithm proposed by Chang [CHA84] to handle letter-oriented keys. This scheme depended on the extraction of unique 2-tuples from the letter-oriented keys. The 2-tuples were extracted from the set of keys by picking characters from two arbitrary character positions in the key.

2.7.2 Matrix Compression

The hashing schemes mentioned above require the use of a matrix compression technique. The n-tuples are extracted from the keys and stored in a matrix. The cardinality of a set is the number of elements in the set. It is a measure of the size of the set. The smaller the cardinality of the extracted n-tuples, the greater the sparseness

of the matrix and hence the more effective are the compression techniques when applied to the matrix. The trial and error schemes used for the extraction of n-tuples in these schemes left the choice of cardinality of the tuples to the discretion of the person making these repeated trials. The proposed genetic scheme will take into consideration the cardinality of the n-tuples as a factor in determining a *fitness function* for the genes.

An efficient implementation of a trie structure using a new internal array structure suggested by Aoe, Mormoto and Sato [AOE92], called the double array, combines the fast access of a matrix form with the compactness of a list form. This paper suggests the possibility of applying the algorithms presented for updating the double array to the reduction of static sparse matrices.

Tarjan and Yao present a scheme for storing a sparse table of n entries [TAR79], each an integer between 0 and N-1, with an access time of $O(\log_n N)$ and a storage of $O(n)$.

2.8 Possibility of a Genetic Solution

The problem of extracting n-tuples that uniquely identify the key space is a combinatorial problem. Genetic algorithms are stochastic search procedures based on the randomized operators, mating, crossover and mutation. Ever since Holland gave the schema analysis [HOL73], genetic algorithms have been successfully applied to solving a wide variety of hard combinatorial problems in areas such as scheduling and transportation problems described by Goldberg [GOL89], partitioning graphs described

by Kernighan [KER70], and the traveling salesman problem described by Lin [LIN65]. The search behavior of the genetic algorithm has been modeled in a Markovian framework by Arunkumar [ARU93] and strong convergence of the search to a solution was proved. Genetic algorithms employ a randomized heuristic search strategy for which a priori complete knowledge of the features of the domain are not required, which makes it favorable for attempting to solve the combinatorial problem of the extraction of n-tuples.

The search strategy adopted in genetic algorithms in the context of combinatorial optimization is described by Arunkumar [ARU93]. The algorithm begins with an initial generation of a uniformly random population of genetic codes that should represent the solution patterns. The solution patterns are the syntactic encoding of a solution. This is an issue of representing the problem in a genetic schema, which in many cases is not trivial, as observed by Michalewicz [MIC92], depending on the nature of the problem. The nature of the problem at hand lends itself easily to a genetic solution because of the ease with which the solution patterns can be constructed, as explained in Chapter 3. Battle and Vose [BAT93] conclude that schemata more general than Holland's can also be made to direct a genetic search, and that a duality exists between problem representations and the schemata that are relevant for their optimization. This duality provides a theoretical framework in which to interpret problem representations.

Holland defined a schema as describing a subset of strings (in a population) with similarities at certain string positions [HOL73]. Vose [VOS91] generalized this notion of the schema in genetic algorithms by defining it as a predicate. Battle and Vose

[BAT93] generalized Holland's schemata in a manner which preserves the algebraic structure, which behaves according to the Schemata Theorem given by Vose [VOS91].

According to Michalewicz [MIC92], the theoretical foundations of genetic algorithms rely on a binary string representation of solutions and on the notion of schema. According to the schema theorem as stated in Chapter 3 of [MIC92], above average solution strings grow exponentially in subsequent generations of the genetic algorithm, resulting in convergence to a solution. Since the nature of the problem being addressed in this paper is such that it favors binary string representations of its solutions, a genetic algorithm designed to solve this problem will guarantee a convergence to a solution, as it is backed up by the Schema theorem.

CHAPTER 3

3. DESIGN AND IMPLEMENTATION ISSUES

This section describes three independent approaches to solving the problem of extracting a set of unique tuples from a given word list. The *combinatorial* and *trial and error* approaches are used as standards to compare and evaluate the performance of the genetic approach, which is the focus of this paper. The genetic algorithm used to solve the problem is implemented by two different approaches: writing a C program and using a genetic application tool called LibGA. This is done in order to investigate the usefulness of the application tool. The results obtained by these two different approaches are compared.

Since the terms *tuple*, *overlap*, and *cardinality* will be referred to on more than one occasion, their definitions are given. *Tuple* is defined as an ordered list of characters. *Cardinality* refers to the length of a tuple and *overlap* refers to the presence of duplicate entries in the list.

3.1 Description of the Genetic Approach

The problem of extracting a unique set of tuples from a given word list involves a search without any prior knowledge about the search domain. A solution is characterized as good according to the following two criteria; low cardinality and low

overlap of the tuples. Since the extracted tuples will be used in a perfect hashing scheme, the solution must put more stress on realizing the latter of the two criteria. The genetic approach to the problem as described in this paper was designed to do just this. Genetic algorithms are well known to be applied to optimization problems. The design of the fitness evaluation scheme, which will be described later in this section, relates directly to the measure of overlap of the tuples. The goal of the genetic algorithm was to effectively search for better candidates for the solution using only the payoff or fitness of each candidate. This search required no auxiliary information in order to work and hence suited this problem very nicely.

3.1.1 Outline of the Genetic Algorithm

The genetic algorithm described in this paper is a classical genetic algorithm, which means that it operates on binary strings. Hence a mapping between the potential solutions and the binary representation is required. This process, called the coding of strings, is explained in detail in section 3.1.2. The genetic algorithm begins with a randomly generated initial population of binary strings. This scheme of forming the initial population is described in section 3.1.3. Then the genetic operators, explained in section 3.1.4, are applied. This application basically involves the copying and swapping of partial strings of the original population to form a new population. The new population thus created is evaluated using an evaluation scheme, explained in detail in section 3.1.5. This process is repeated as shown in Figure 5, until a termination is reached. The

conditions leading to a termination of the cycle are explained in section 3.1.7.

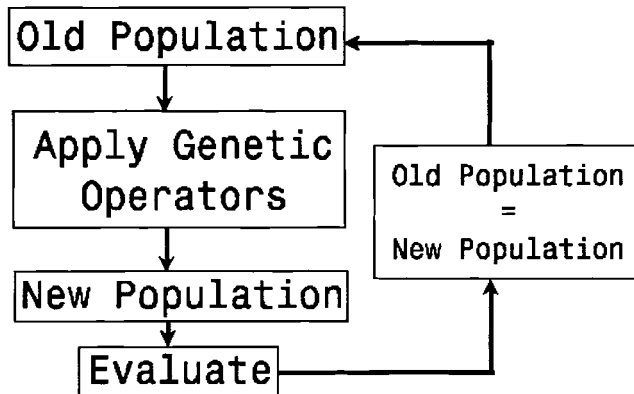


Figure 5: Outline of the genetic algorithm

During the genetic cycle some of the chromosomes are just copied into the new population and others are modified by the genetic operators and then introduced into the new population. When implementing a fixed size population it is required to replace existing chromosomes by the new modified ones. Different implementations can use different techniques of performing this replacement process. In the genetic program described by Goldberg[GOL89], two chromosomes selected for mating are invariantly replaced by their resulting offspring. In the application tool called libGA, which is discussed in section 3.1.8.2, the replacement strategy is different. Of the four chromosomes involved in the mating operation, i.e. the two parents and their two offspring, only the best two chromosomes are selected for the new population.

3.1.2 Coding of Strings

The coding of strings that represent potential solutions facilitates the mapping of the parameters of the optimization problem to a string in the population. The parameters of optimization are the tuple length and the amount of overlap. Thus a string should map to a list of tuples of a known length and overlap. Consider the binary string 01101. In this string the 1's occur at the character positions 2, 3, and 5. The tuples that correspond to this string are obtained by picking out the characters at these positions from each word in the word list. For example, Figure 6 shows the mapping of the binary string 01101 to a set of tuples for a given word list. In general, for a binary string $x_1x_2x_3\dots x_n$ where x_i is either a 0 or 1 for all i in $(1,n)$ and n is the length of the words in the data set, the tuples that this string represents are obtained by picking characters from each word at positions i for which $x_i = 1$. The length of the tuple is the number of 1's in the string. The measure of overlap is represented by the *hashlength*, which is computed from the tuples that the string represents.

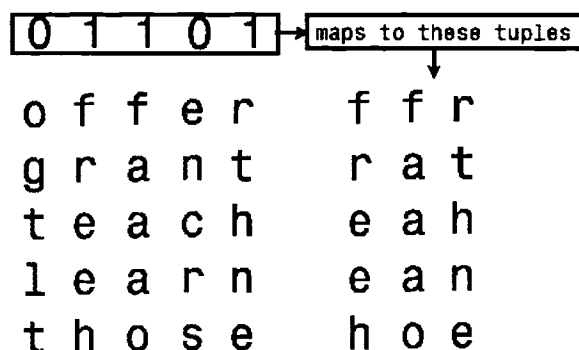


Figure 6 : Genetic code

3.1.3 Creation of the Initial Population

The population size is stored in the variable *popsiz*e, which is input to the program during execution. The initial population contains chromosomes selected from the solution space. The complete solution space consists of $2^n - 1$ chromosomes, where n is the word length of the words in the input list. For example, if we consider words of length 3 then the complete solution space consists of the chromosomes, 001,010,011,100,101,110,111. In general, the solution space is much larger than the population size. Hence the initial population can contain only a few chromosomes from the solution space. This selection process is implemented using two methods: a random initialization, and a biased initialization.

3.1.3.1 Random Initialization

The initial population contains chromosomes selected randomly from the entire solution space. The logic for creation of the initial population follows,

for $i = 1$ to *popsiz*e

$x =$ a random number in the range $[1..2^n - 1]$

chrom = a binary string equivalent of x

insert *chrom* into the population

end loop

If the population size is smaller than or equal to the solution space, then the initial

population is constructed to contain the whole solution space.

3.1.3.2 Biased Initialization

In the biased initialization method, the initial population will consist of k_1 number of 1-tuples, k_2 number of 2-tuples, k_3 number of 3-tuples, ..., k_n number of n -tuples. The values $k_1, k_2, k_3, \dots, k_n$ are computed from the following equation:

$$k_i = \frac{p_i}{\sum_{i=1}^n p_i} \cdot ps, \text{ where } p_i = \frac{r^n - r^{n-i}}{r^n - 1}$$

, where ps is the population size, p_i is the probability that two tuples of size i will not be identical, r is the alphabet size, n is the word length of the input word list. The expression for p_i is derived below.

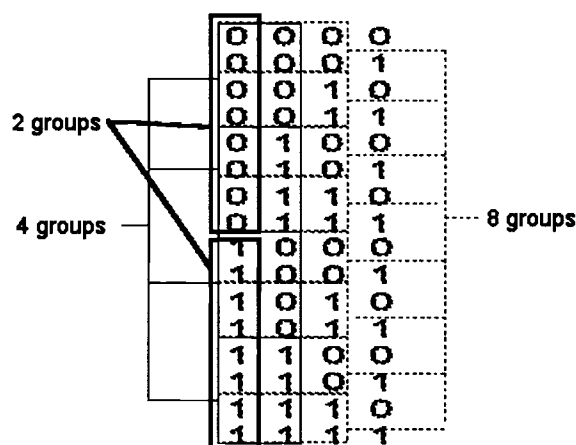


Figure 7: A sample solution space

Consider the simple case of an input word set consisting of words of length 4, formed from the alphabets: 0,1. There are altogether 2^4 possible words that can be formed from this alphabet which constitutes the solution space, as shown in Figure 7. There are 2^1 possible ways of constructing tuples of size 1. However there are totally 2^4 tuples. Therefore, the 2^4 1-tuples consists of 2 groups of $2^4/2^1 (= 8)$ identical 1-tuples, as seen in Figure 7. Similarly it can be said that the $2^4 = 16$ possible words will consist of

4 groups of $2^4/ 2^2 (=4)$ identical 2-tuples

8 groups of $2^4/ 2^3 (=2)$ identical 3-tuples

1 group of $2^4/ 2^4 (=1)$ identical 4-tuples

3.1.3.2.1 Probability that any two 1-tuples will be identical The total

number of ways of selecting 2 1-tuples from a list of 16 =

$${}^16C_2 = 120$$

The number of ways of selecting 2 0's =

$${}^8C_2 = 28$$

,since there are 8 0's altogether in the list of 1-tuples .

Similarly the number of ways of selecting 2 1's from a group of 8 1's is also = 28

Therefore the probability of two 1-tuples being identical =

$$\frac{\text{number of ways of selecting 2 0's} + \text{number of ways of selecting 2 1's}}{\text{Total number of ways of selecting two 1-tuples}} =$$

$$\frac{{}^8_2 C_2}{{}^{16}_2 C_2} = \frac{28 \cdot 2}{120} = \frac{56}{120}$$

3.1.3.2.2 Probability that any two 2-tuples will be identical Since there are 4 groups of 4 identical 2-tuples, by a similar treatise it can be seen that this probability evaluates to :

$$\frac{{}^4_4 C_2}{{}^{16}_2 C_2} = \frac{6 \cdot 4}{120} = \frac{24}{120}$$

3.1.3.2.3 A Generalization : Probability that any two k-tuples will be identical

For a list of words of word length n and alphabet size r , the total number of distinct tuples that can be formed is r^n . This is the size of the entire solution domain. Following along the line of intuition presented in the previous two sections, it can be seen that this entire solution domain of size r^n will contain r^k groups of r^{n-k} identical tuples. The probability of two k -tuples being identical, $1 \leq k \leq n$, is given by :

(number of groups of identical k-tuples)(probability of selecting two tuples from a group)
 Total number of ways of selecting two tuples

$$= \frac{r^k \cdot \binom{r^{n-k}}{2}}{\binom{r^n}{2}} = \frac{r^{n-k}-1}{r^n-1}$$

The probability that two k-tuples will be different :

$$1 - \frac{r^{n-k}-1}{r^n-1} = \frac{r^n-r^{n-k}}{r^n-1}$$

3.1.4 The Genetic Operators: Reproduction, Crossover, and Mutation

The functioning of each of the three genetic operators depends on random choice. This random choice is implemented in three functions : *Random*, *Rnd*, and *flip*. The first of these returns a pseudorandom number between 0 and 1. *Rnd* returns an integer value between specified lower and upper limits, and *flip* returns a boolean true value with specified probability.

In the genetic algorithm, reproduction is implemented in the function *select* as a linear search through a roulette wheel with slots weighted in proportion to string fitness

values. *Select* returns the population index value corresponding to the selected individual.

A routine crossover takes two parent strings called *parent1* and *parent2* and generates two offspring strings called *child1* and *child2*. The probabilities of crossover and mutation, *pcross* and *pmutation*, are passed to crossover, along with the string length *lchrom*. At the beginning of the routine crossover, function *flip* is used to determine whether the crossover operation will be performed on the current pair of parent chromosomes. The function *flip* uses *pcross* as an argument to determine whether a cross is called for. The crossing site is selected randomly using the function *Rnd*, which returns a pseudorandom integer between 1 and *lchrom*. If no cross is to be performed, then the mutation operator visits each bit and calls the flip function with argument *pmutation* to determine whether that bit value will be altered.

3.1.5 The Evaluation Scheme

Literature confirms that there is no deterministic method other than an exhaustive search to solve this problem. Although genetic algorithms use probabilistic transition rules to guide their search, this use does not suggest that the search is some simple random search. The Genetic algorithm uses the fitness function as a tool to guide the random search toward regions of the search space with likely improvements. The genetic approach was designed with the prospect of achieving better performance than a totally random approach (trial and error) at one extreme and a totally deterministic approach

(exhaustive search) at the other.

The population is a collection of candidate solutions that are represented by binary strings. Each binary string represents a set of tuples. The mapping of a binary string to the set of tuples that it represents is accomplished easily by picking characters from the word list at positions corresponding to the 1's in the binary string. For example, consider the list of words shown in Figure 6. The highlighted characters of the word correspond to the 1's in the binary string. These characters form the tuples that the binary string represents, namely {ffr, rat, eah, aan, hoe}. The evaluation scheme assigns a fitness value to each binary string by computing the *measure of overlap* present in the set of tuples that are mapped to it. This *measure of overlap*, also called *hashlength*, is computed as follows. Consider an individual of a population represented by the binary string 01011. Let the input word list be stored in an array of strings W_N , where N is the total number of words in the list. Then the set of tuples that are mapped to the binary string 01011, obtained by extracting characters from positions 2,4, and 5, are denoted by the list :

$$\begin{aligned} T_1 &= (W_{12}, W_{14}, W_{15}) \\ T_2 &= (W_{22}, W_{24}, W_{25}) \\ &\cdot \\ &\cdot \\ T_N &= (W_{N2}, W_{N4}, W_{N5}) \end{aligned}$$

By way of frequency count, this set of N tuples $T = \{T_1, T_2, \dots, T_N\}$ can be partitioned into a set of groups $G = \{G_1, G_2, \dots, G_w\}$ with the following properties :

- Each group G_i is a set of one or more tuples.
- If $G_i = \{t_1, t_2, t_3, \dots, t_k\}$ a set of k tuples, then $t_1 = t_2 = t_3 = \dots = t_k$. i.e.

all the tuples in each group are identical.

- . If $i \neq j$, then $G_i \cap G_j = \Phi$, the null set.
- . If $\text{Ord}(X)$ represents the number of elements in the set X , then

$$\sum_1^W \text{Ord}(G_i) = \text{Ord}(T) = N$$

When all the tuples are unique, as seen from the above equation, $W = N$ which means that each group will contain only one tuple.

The hashlength of the set of tuples T , corresponding to the binary string is evaluated by the summation :

$$\text{hashlength}(T) = \sum_{j=1}^W \sum_{k=1}^{\text{Ord}(G_j)} k$$

When there is no overlap among the tuples, all the tuples are unique and $W = N$ and $\text{Ord}(G_j) = 1$, resulting in $\text{hashlength} = N$, the number of words in the input word list. The fitness function for this case must be evaluated to its maximum value, which is 1. Thus the fitness function is computed as :

$$\text{fitness}(T) = \frac{N}{\text{hashlength}(T)}$$

When there is an overlap, the $\text{hashlength} > N$ and the fitness evaluates to a fraction. The smaller this fraction is, the higher the overlap.

The crossover/mating operation is designed to achieve a guided search toward regions of the search space with likely improvement as illustrated in the following

scenario. Consider the following two individuals/chromosomes that may have been selected for mating: 101100 and 010011. After the mating operation, the overlap of the offspring may or may not be significantly less than that of the parents. Upon crossover the best parts or contributors to fitness of each chromosome may be combined into one, resulting in a higher fitness value of one of the offspring. This scenario may not take place, but when it does, the search moves significantly towards a better solution.

The evaluation of the fitness is based solely on the computation of the hashlength according to the above equation for *fitness()*. The hashlength depends directly on the number of duplicate entries among the list of tuples. The higher the number of duplicates, the greater the value of the hashlength. It is clear that choosing more letters from the word list to form the tuples will not make the hashlength worse. Stated in another way, if T1 is the set of tuples mapped by the chromosome 011000 and T2 is the set of tuples mapped by the chromosome 011011 then the following is always true:

$$\text{hashlength}(T1) \leq \text{hashlength}(T2)$$

This happens because the latter of the two chromosomes considers two additional character positions to form the tuple T2. If *hashlength*(T2) is greater than *hashlength*(T1) then the tuple set T2 should be given a greater fitness value than T1. This is indeed the case in the evaluation scheme. But in the case where the two hashlengths are identical, the tuple set T1 should be given a greater fitness value because the tuple size for T1 is less than that for T2. The evaluation scheme does not recognize this situation and ends up assigning a higher fitness to T2. This situation is rectified by a repair algorithm that keeps the cardinality, or tuple size, under check. This algorithm

is explained in section 3.1.3.

The genetic operators are applied to selected candidates of the population and the resulting offspring are introduced into the new population. The probability of selection of a candidate is designed to be proportional to its fitness. This selection is repeated until the new population reaches a desired size, which is at least the size of the old population. As this process is repeated again and again, duplicate copies of a particular candidate will dominate the population. Theoretically this leads to a complete domination by the individual. At this point the genetic algorithm is terminated and the solution is represented by this dominant individual.

3.1.6 Enforcement of the Constraints

The evaluation scheme as it stands serves the purpose of minimizing only the overlap present in the extracted tuples. The genetic search must incorporate the constraint on the length of the tuple; otherwise the search suffers a potential risk of favoring tuples of high cardinality and ending up with the redundant solution of a tuple set that is identical to the word list itself. The constraint must be enforced to ensure that the search will be directed towards solutions whose combination of overlap and tuple-length is minimal. This is stated more formally as follows: Let P denote a population of size n . Then P_i denotes the i^{th} individual of the population P . $F(P_i)$, $O(P_i)$, and $L(P_i)$ denote the fitness, a measure of overlap, and the length of the tuples that are represented by the individual P_i . For all i in $\{1, n\}$ $F(P_i) <$ every element in the set $S : \{F(P_j) \mid$

$O(P_j) = O(P_i) \ \& \ L(P_j) > L(P_i), \ 1 < j < n \ \text{and} \ j \neq i$. In the implementation, the elements of the set S are assigned a very low fitness value so that they eventually die out. These candidates are undesirable since they violate the tuple-length constraint.

The implementation of this constraint is explained. A table called *RestrainLength* is used to update the smallest hashlength for each tuple size. For example, consider all tuple sets of size one extracted from a list of words of length four. There are four such tuple sets obtained by string coding the four chromosomes, 1000, 0100, 0010, 0001. The process of string coding was explained earlier. As each of these four chromosomes are evaluated in the course of the genetic search, the first entry of the table *RestrainLength* will be updated to the smallest hashlength. This update algorithm is described below:

chrom :represents the chromosome being evaluated
 card (chrom) :represents the cardinality of the chromosome *chrom*
 hashlength (chrom) :represents the hashlength of the tuples mapped by the
 chromosome *chrom*

if hashlength(chrom) < RestrainLength(card(chrom)) *then*

RestrainLength(card(chrom)) = hashlength(chrom)

If any of the entries above the updated entry has a hashlength that is less than or equal to the hashlength of the updated entry, then the chromosome corresponding to the updated entry is undesirable and assigned a low fitness value.

3.1.7 Conditions leading to Termination of the Genetic Algorithm

The genetic algorithm forms a new population from the existing one on each iteration of the genetic cycle. The population created on each cycle is known as a generation. Theoretically the population representing the n^{th} generation as n approaches infinity will contain chromosomes that are perfect solutions to the problem. In practice, however, the cycle is terminated when one of the following two conditions occur.

1. *A solution is found.*

This occurs when all the individuals of the population attain the same fitness value, and is referred to as *convergence to a solution*. This does not necessarily mean that all the individuals of the population are identical. The genetic algorithm may find more than one solution, if it exists. Elitism is used to improve the performance of the genetic algorithm by speeding up the rate of convergence. Use of elitism ensures that each successive generation is as good or better than the generations that precede it. This is implemented by maintaining the best few chromosomes in each generation. Occasionally the application of genetic operators on good chromosomes yield poor successors, resulting in a drop in the fitness of the succeeding generation. Use of elitism ensures a monotonic increase in the fitness of a generation because the best few chromosomes are reintroduced into the population. Implementation of elitism in the C program implementation is carried out by replacing the parents by their offspring only if the offspring are better.

2. *A solution is not found.*

A counter called *gencount* is used to keep track of the number of generations created. When *gencount* exceeds *MAXGEN*, a preset constant, the cycle is terminated. This indicates that a solution was not found within the maximum number of iterations allowed.

3.1.8 Implementation Issues

This section describes the two approaches used to implement the genetic algorithm. In the first approach, a C program is written to implement all the aspects of the genetic algorithm. The code is written entirely from a scratch and includes the design and implementation of the data structures, the code for the genetic operators, the code for the evaluation scheme, the code for various randomized operators, and other support modules. In the second approach use of a genetic application package helps to drastically reduce the amount of coding. Only the code for the evaluation scheme and the main driver routine has to be constructed.

3.1.8.1 Approach 1: The C program

The implementations of the data structure for the population and the genetic operators are identical to the implementations presented by Goldberg [GOL89]. The source code for the genetic algorithm is written in the 'C' programming language. The choice of a binary string as a data structure for the chromosome well suited the

transformation of the problem into genetic coding as explained in the section on solution coding. In the genetic algorithm the probability of mutation *pmutation* is set at a much lower value than the probability of crossover *pcross*. Mutation by itself is a random walk through the string space. When used sparingly with reproduction and crossover, it prevents the premature convergence to a non-optimal solution. This is due to the fact that *the potentially better solutions* that cannot be generated by recombination of existing ones will be introduced by use of the *mutation* operator, thus preventing convergence to an otherwise optimal solution.

The actual population size is set as an input parameter to the genetic algorithm, so that its value can be altered easily. The strength of the genetic approach comes from the fact that the population size is usually much smaller than the actual solution domain. The genetic search focusses on the more promising parts of the solution space instead of on the entire solution space. As an input parameter, the population size can be set at different values for different data sets to study its effects on the convergence of the genetic algorithm.

3.1.8.1.1 Data Structures The primary data structure for the genetic algorithm is a string population. The population is constructed as an array of individuals where each individual contains the genotype (the artificial chromosome or bit string), and the fitness (evaluation function) value along with other auxiliary information such as the cross point and the parents involved in the mating. A number of constants are defined: the maximum population size, *MAXPOP*, and the maximum string length, *MXLNGTH*.

These set upper bounds on the population size and the string length. The type *population* is an *array* of type *individual*, which is indexed between 0 and *MAXPOP-1*. Type *individual* is a record composed of a type *Chromosome* called *chrom*, and a real variable called *fitness*. These represent the artificial chromosome, and the string fitness value. Type *chromosome* is itself an array of type *gene*, which is indexed between 0 and *MXLNGTH-1*. *Gene* is another name for the character type to represent a single gene value of '1' or '0'.

In the genetic algorithm, the genetic operators are applied to the entire population in each generation. In the implementation, the offspring of the genetic operations replace the parents in the new population. The population size remains fixed. The computer implementation uses a number of important global variables: *pcross*, *pmutation*, and *sumfitness*. The variables *pcross* and *pmutation* are the probabilities of crossover and mutation respectively. The *sumfitness* variable is the sum of the population fitness values.

The array *RestrainLength* of size *MXLNGTH* is defined to be of type long. This array is used to update the maximum fitness at each tuple length that accumulates during the genetic cycle. The information in this array is used in the evaluation function to keep the tuple length under check. This is explained in detail in a later section.

3.1.8.1.2 The Driver/Main program The driver/main section of the program serves as a breeding ground for each generation to evolve into the next through a repeated cycle or loop. The generation counter is incremented, the function *generation*

is called to generate a new generation, and the population is advanced: *oldpop* = *newpop*. In addition, the main program is responsible for reading in the input data for the problem, randomly initializing the first population, calculating of relevant statistics, and reporting various parameters for plotting graphs. As part of the statistics, the best *k* strings accumulated over every three generations are reintroduced into every fourth generation. This procedure, called elitism, ensures the monotonic increase in the average fitness level of the population over the generations. The value of *k* is selected as the bigger of 1 and $\lfloor \text{population_size}/25 \rfloor$.

3.1.8.2 Approach 2: Using the LibGA application tool

When using the LibGA application tool [COR93], only the code for the evaluation scheme and the main driver routine need to be written. The code for the evaluation scheme is written in a function called *obj_fun()* which is declared in the main program *main()*. This main program consists of essentially two parts, namely, the initialization and the execution of the genetic algorithm. The *obj_fun* takes a chromosome as an argument and returns a real fitness value. The chromosome is defined as a string of bits using a configuration file. The process of evaluating the fitness of a chromosome remains the same as before. The only difference lies in the coding of the main program, the configuration file, and the user data file. These are explained below.

3.1.8.2.1 The Main program The main program consists of two parts, the

initialization and the execution. The initialization of the genetic algorithm is accomplished by the function *GA_config* ("*tupleapp.cfg*", *obj_fun*). This function sets the configuration of the genetic algorithm to the contents of the file *tupleapp.cfg* and sets the function *obj_fun* as the evaluation scheme for the genetic algorithm. The function *obj_fun* takes a data structure of type *Chrom_Ptr* as the argument. This data structure is shown in Appendix A. The chromosome is stored in an array of type *Gene_Ptr*. Other useful information is stored along with the chromosome, such as its length and fitness. A population consists of a group of chromosomes. So the data structure for the chromosome contains an index that is used to reference this chromosome in the group. The configuration of the genetic algorithm is stored in a data structure of type *GA_Info_Ptr*, which is returned by the function *GA_config*. The configuration file contains the settings for the data type of the gene, the chromosome length, the population size, the selection method, the genetic algorithm type, the crossover method, the mutation method, etc. The configuration is set to a default setting unless changed. In the configuration file *tupleapp.cfg*, the following settings are used:

<i>user_data</i>	the name of the input data file that contains the word list
<i>data_type</i>	data type of a gene is a <i>bit</i> representing 1 or 0
<i>initpool</i>	random (initial population is generated at random)
<i>length</i>	length of the chromosome is set from the input data file
<i>pool_size</i>	the population size is set from the input data file

The remaining configuration is at the default settings.

Every application will have its own configuration file where the configuration settings

for that application are stored. The configuration file used for this application is called *tupleapp.cfg*. As mentioned earlier, one of the settings called *user_data* is used to store the name of the input data file. This input data file will contain the word list from which the tuples will be extracted. A function called *readwords ()* reads the input word list from the data file *user_data*. The execution of the genetic cycle is started by simply calling the function *GA_run (ga_info)*. The argument *ga_info* is a data structure of type *GA_Info_Ptr* that is initialized to the contents of the configuration file.

3.1.8.2.2 The Configuration file The configuration file consists of settings that are used to initialize the data structure *GA_Info_Ptr*. The components of this data structure are given in Appendix A. Some of the components are explained briefly. The entry *user_data* is used to store the name of the input data file. *Data_type* is an integer flag representing the data type of the gene. The different data types allowed for a gene are, bit, int, and real. The entry *chrom_len* refers to the length of the chromosome. A boolean *minimize* is used to represent the type of optimization that is needed. If *minimize* is *true* then the genetic algorithm minimizes the *obj_fun*; otherwise the *obj_fun* is maximized. The crossover rate and the mutation rate are contained in the entries *x_rate* and *mu_rate* respectively.

3.1.8.2.3 The User Data file The user data file is the file that contains the input word list. The first line of this file contains the number of words, the length of each word, and the alphabets used to form the words. This is followed by the word list. The

function *readwords* reads the contents of this file and stores the words in an array of type string. In the evaluation of the fitness of a chromosome, the tuples that it represents are extracted from these words. The evaluation scheme, as explained in section 3.1.5 takes these tuples as input to determine the fitness of the chromosome.

3.2 Description of the Combinatorial/Exhaustive Search Approach

The combinatorial approach systematically eliminates all possible non-solutions starting from tuples of cardinality 1 and ending at the whole word length. At any point in this process as soon as a solution is found the program terminates immediately. A solution is one for which the overlap is 0. A non-solution is one for which an overlap exists. The extent of this overlap is not evaluated as it serves no purpose to this search method in yielding a solution. All possible combinations of tuples of size 1 are tested first. Then all combinations of tuples of size 2 are tested and so on until tuples of size w are tested, where w is the length of the whole word. Thus in the worst case the number of trials made computes to :

$$C_1^w + C_2^w + \dots + C_w^w$$

where w is the length of the word. This approach guarantees finding an optimal solution if one exists.

When the number of words and the alphabet size used to form the words are known in advance, the combinatorial approach can be modified to improve its performance. Consider a list of words constructed from an alphabet of size r . The total

number of unique tuples of size 2 that can be constructed is r^2 . Hence if the total number of words in the list is greater than r^2 , any set of tuples of size 2 extracted from the list will have duplicates. So any tuple of size 2 cannot be a solution. In general any tuple of size k , for which $r^k <$ total number of words in the input list, is not a solution and can be skipped. This technique is incorporated into the combinatorial algorithm to improve the efficiency of the exhaustive search.

3.3 General Description of the Trial and Error Approach

The trial and error approach to finding a solution is a completely random process in which the solution is sought by repeated attempts or guesses at the tuples hoping to hit the mark. This approach does not guarantee a solution even if one exists. Also the solution, if found, is not necessarily the optimal one.

The basic technique of the trial and error approach consists of picking characters at random positions from each word in the word list. Each such attempt, called a trial, can be represented as a sequence of character positions. For example, one such trial sequence $[3,5,4]$ represents the set of tuples of size 3 obtained by extracting characters from positions 3, 4 and 5 from the word list. The tuples obtained on each trial are evaluated to determine the overlap. If the overlap is 1 then the solution is found and no more trials are made; otherwise, another trial to guess the solution is made. No assumption or knowledge of the word list is used in taking a guess at the solution. A specified upper limit on the number of trials on each tuple size is used. When this limit

is exceeded then trial sequences of the next larger size are generated and so on. This process is repeated until either a solution is found or the limit on the total number of trials allowed is exceeded.

3.4 About the Implementation Platform

The machine used is a Sequent Symmetry S81 with 24 80386 processors running at 20 MHz each and with a RAM of 108 Mbytes. Each processor also has both a 80387 and a Weitek floating point co-processor.

CHAPTER 4

4. RESULTS AND OBSERVATIONS

The problem is to extract a unique set of tuples from a given word list. A tuple is a sequence of characters picked from a word in the list. All the tuples are extracted by picking characters in the same sequence from each word in the list. The tuples must not only be distinct but also obtained by picking as few characters as possible. A genetic algorithm is implemented to solve this problem. The results obtained by this algorithm are displayed in the form of graphs. From these results the performance of the genetic algorithm is observed. The input data consists of a list of words. Each word in the list is constructed by randomly selecting characters from a given alphabet set. For the purpose of evaluation of the genetic algorithm, the length of the words and the number of words in a data set are varied.

The genetic algorithm implemented using the C programming language is called *gengrph*. Another implementation called the *tuple-app* uses the libGA application tool. The results obtained by these two implementations are compared in section 4.5. The performance of the genetic algorithm *gengrph* is compared to the combinatorial algorithm *combon*, that uses an exhaustive search technique to find an optimal solution to the problem. This is discussed in sections 4.2 and 4.3. Also the effect of using a biased initialization of the initial population, based on the probabilities established in section 3.1.3.2 of chapter 3, on the performance is observed in section 4.4.

4.1 The Performance of the Genetic Algorithm

The performance of the genetic algorithm is evaluated in terms of the execution time required to find a solution and the quality of the solution found. The execution time is measured in microseconds and represents the *cpu* time required to find a solution. The *combon* uses an exhaustive search and guarantees to find an optimal solution. The solution obtained by the genetic algorithm is optimal if the solution's fitness is 1.00 and the size of the solution tuple is same as that of the solution obtained by *combon*. The two types of graphs presented in the following sections display results of comparing the execution time and solution quality obtained by both the genetic and combinatorial algorithms.

4.2 The Execution Time

The execution time required to find a solution is compared for both the methods: *combon* and *genrph*. The solution obtained by the combinatorial algorithm *combon*, is guaranteed to be optimal but the exhaustive search that it employs is time consuming. The performance of *genrph* is studied to determine under what circumstances it will outperform the combinatorial algorithm *combon*. This section discusses the execution times, but the quality of the solutions obtained from these runs are discussed in the next section.

The execution time plotted on the y-axis is representative of an average over

several runs. On each of these runs a different set of words of the same length and number is used. The execution times for all these runs are averaged. The graphs that depict this performance use two types of x-axes: one in which the word length is fixed and the number of words varied, the other in which the number of words is fixed and the word length varied.

For the graph shown in Figure 8, the size of the input data set increases along the x-axis. For all the data sets on the x-axis the word length remains fixed at 6. Each point on the y-axis is computed as an average of the execution times obtained on 40 separate runs. For example consider the fourth point on the *genrph* curve.

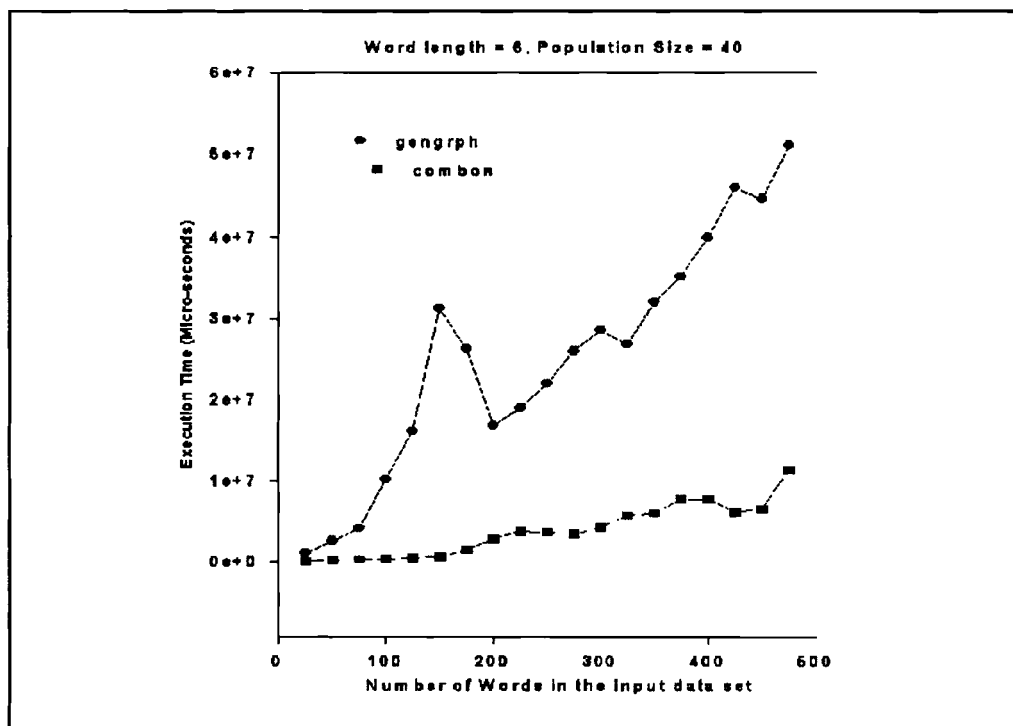


Figure 8: Performance Graph for words of length 6

The x-coordinate of this point corresponds to the input data containing 100 words. The

y-coordinate of this point was obtained by generating 40 different data sets each containing a 100 words and taking the average of the execution times for each one of these data sets. The graphs shown in Figures 9 and 10 are constructed in exactly the same fashion, except that the word length used in Figure 9 is 8 and that used in Figure 10 is 10. The graph in Figure 8 shows that the *combon* method outperforms the *gengrph*. We also see from Figures 8, 9, and 10 that as the number of words in the

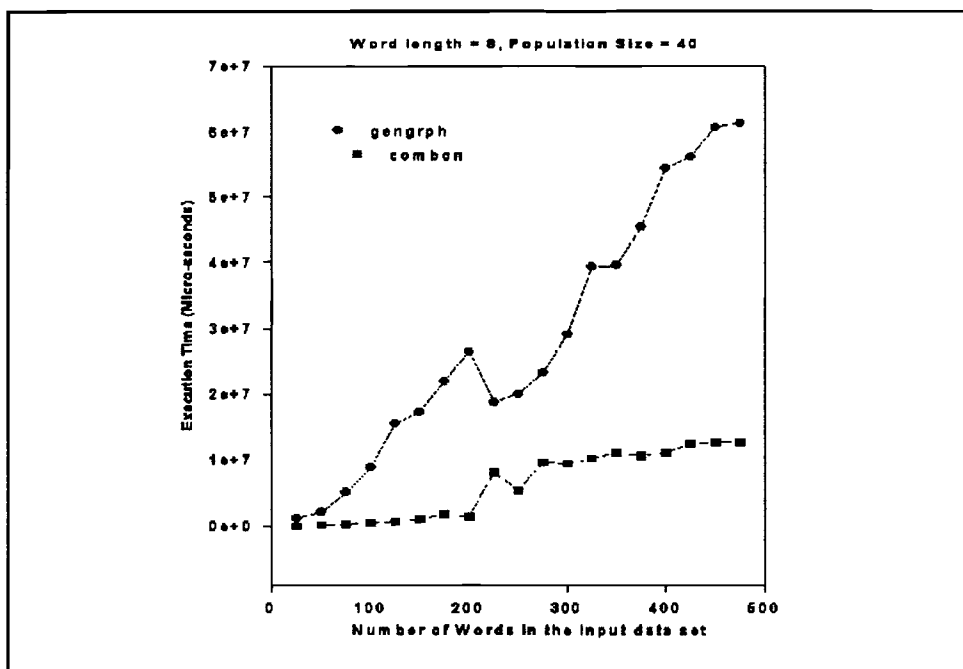


Figure 9: Performance graph for words of length 8

input list increases, the *gengrph* curve rises more rapidly than the *combon* curve.

From Figure 8 to Figure 9, the gap between the performance of the two methods has reduced. From Figure 9 to Figure 10, the gap has reduced even more. From this observation, it seems that although the execution time of both the methods increase linearly with input data size, the increase in word length favors the performance of the

genetic method *gengrph*. It was noted in section 3.1.3 of Chapter 3 that the size of the entire solution domain increases exponentially with word length. Since the *combon* uses an exhaustive search mechanism, it comes as no surprise to see why the performance of the genetic algorithm *gengrph* is rapidly catching up to the performance of the *combon*. The genetic search is a biased form of the trial and error search because more trials are made in regions of the search space that look promising than in those that are not.

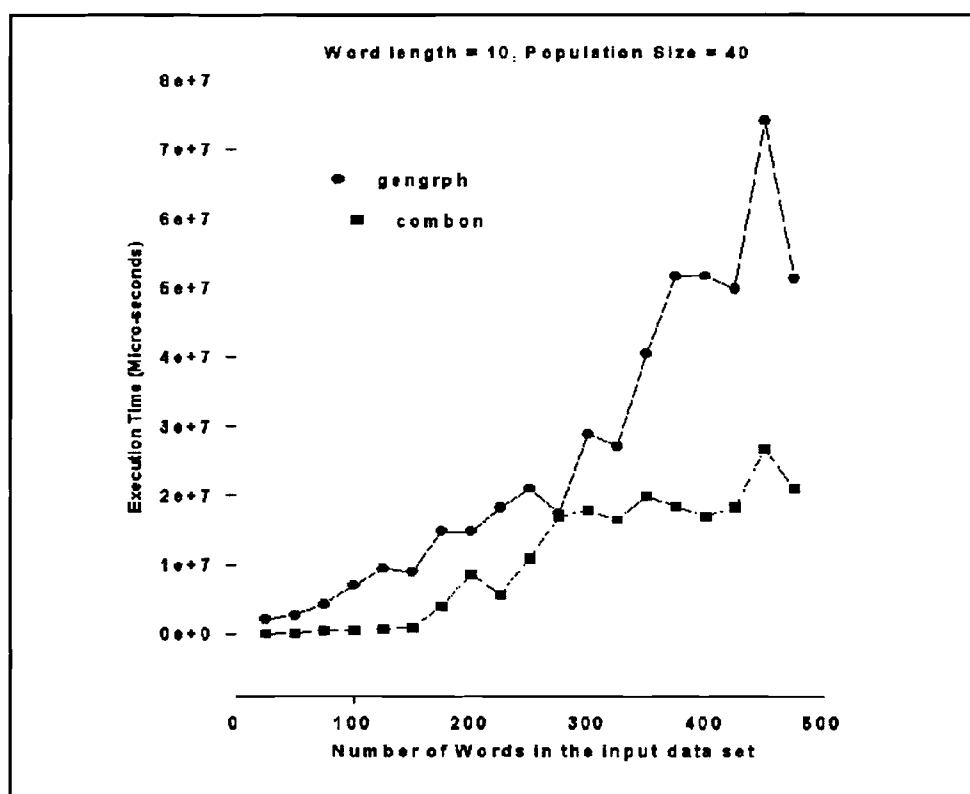


Figure 10: Performance graph for words of length 10

Thus the increase in the word length does not seem to effect the performance of *gengrph* as much as it effects *combon*. This phenomenon is seen in the graph shown in Figure 11. The graph in this figure uses a varying word length along the x-axis, and a fixed

size for number of words in the input data set. As before, each point on the y-axis is computed as an average over 40 separate runs. From this graph, a marked difference in the performance of the two methods is seen. The *gengrph* appears to outperform the *combon* after a certain word length is crossed. The *combon* shows an exponential rate of growth of the execution time, where as the genetic algorithm remains almost unaffected. The graph in Figure 12 is constructed in a similar fashion using input words of increasing length, but the number of words used in all the data sets is now fixed at 600. The *gengrph* curve in this graph exhibits the same form as before, running almost

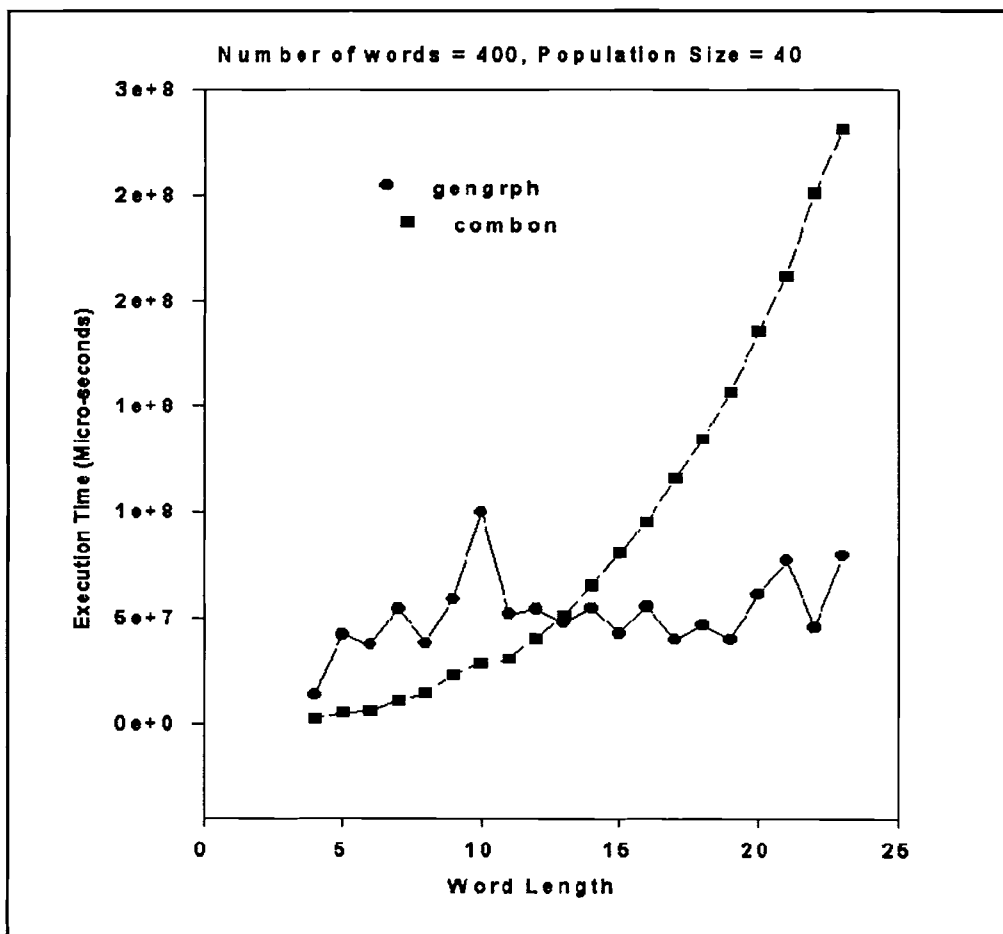


Figure 11: Performance graph #1 for words of increasing length

level in a zigzag fashion across the figure. The combon curve rises exponentially across the figure, as expected. The genetic algorithm outperforms the combinatorial algorithm when words of larger length are considered.

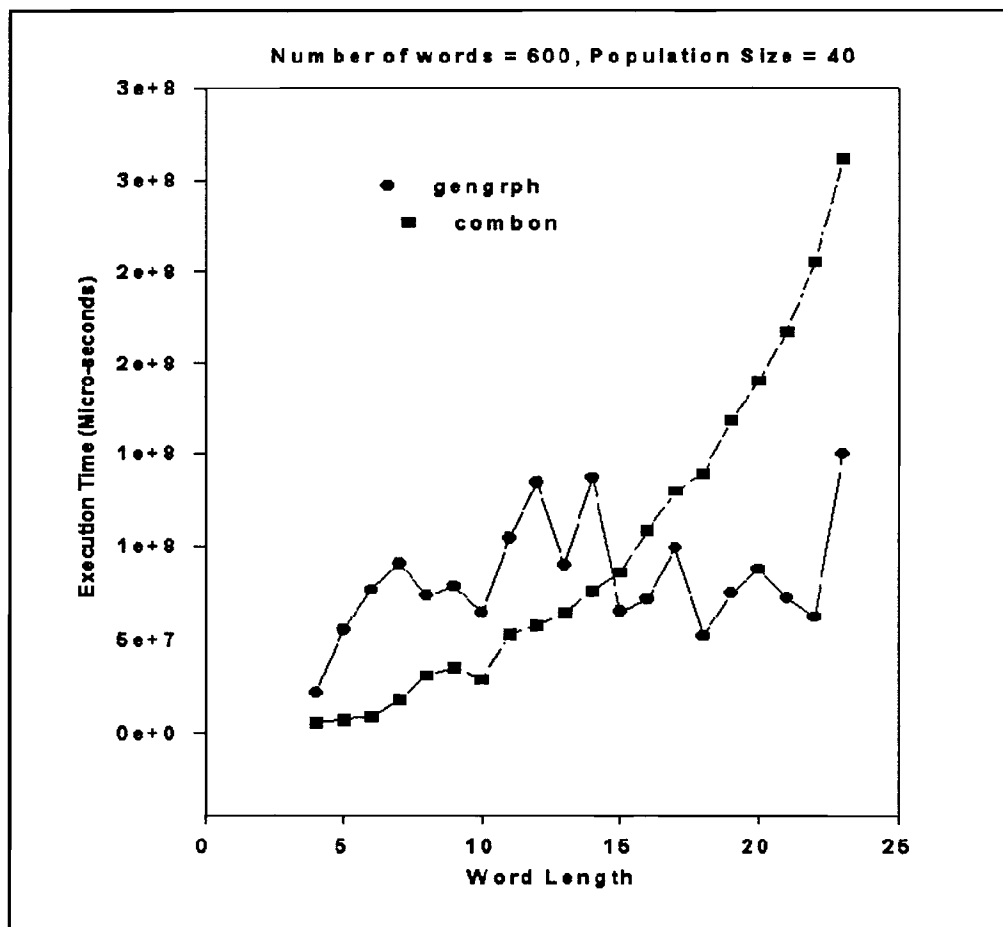


Figure 12: Performance graph #2 for words of increasing length

It is not enough to note a significant improvement of the performance relating to execution time. The question is whether the genetic algorithm finds acceptable solutions to the problem with this efficiency. The quality of the solutions obtained by the genetic algorithm is discussed in the next section.

4.3 The Solution Quality

The quality of the solution is measured in terms of solution tuple size and the solution tuple's fitness. The quality of the solutions obtained by the genetic algorithm is determined by comparing it with the optimal solutions obtained by the combinatorial algorithm.

The graph in Figure 13 shows the sizes of the solution tuples obtained by both the methods: *gengrph*, and *comben*. These solutions were obtained in the execution times

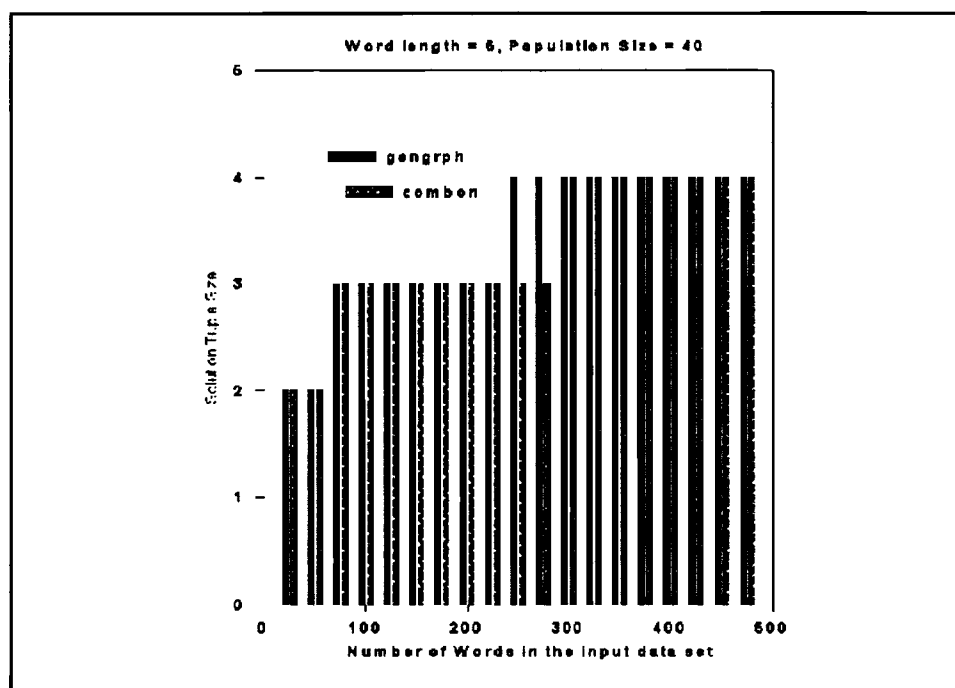


Figure 13: Solution quality for words of length 6

represented in Figure 8. The graph has the same x-axis as in the graph of Figure 8, but the y-axis represents the average of the solution tuple sizes obtained on 40 separate runs

for each input data set on the x-axis. It is seen in the graph that the optimal solution

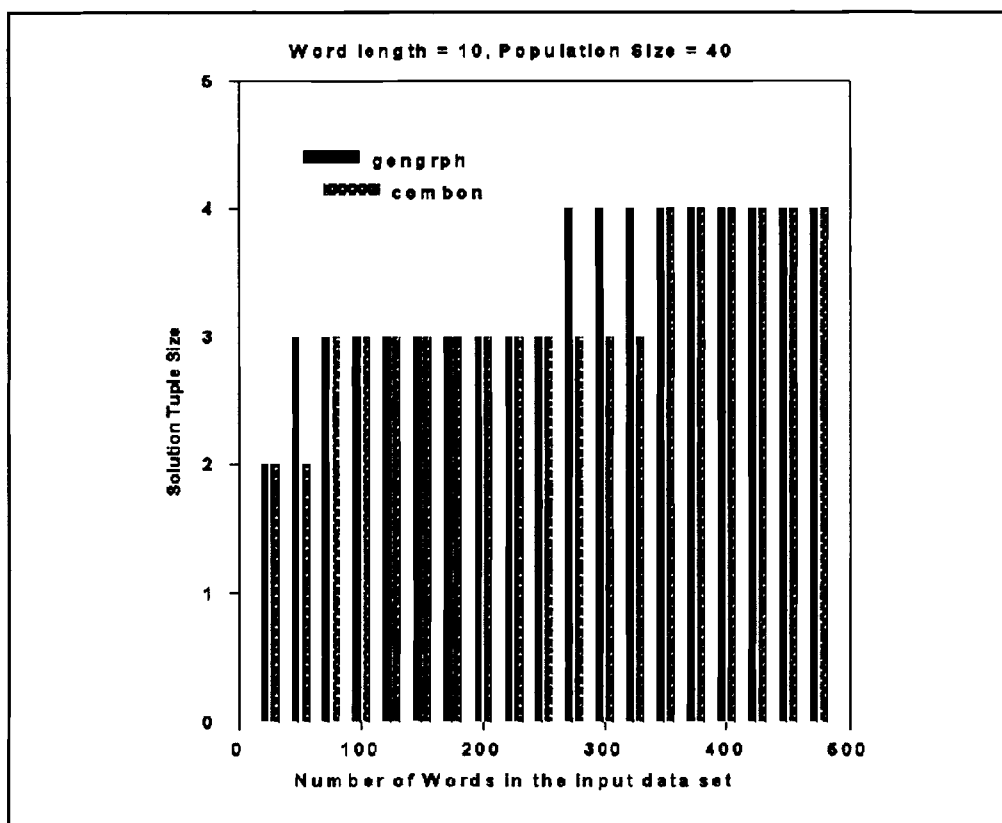


Figure 14: Solution quality for words of length 10

tuple size seems to increase with the size of the input data set. This is expected, since more the number of words, the higher the chance of overlap among the extracted tuples. Hence tuples must use more characters to avoid overlap, resulting in an increase of the solution tuple size. The graph in Figure 14 is constructed in the same fashion except that all the data sets on x-axis now use a larger word length of 10. A match between the size of the solution tuples obtained by *gengrph* and *combon* indicates that the genetic algorithm found an optimal solution. By comparing the graphs of Figures 13 and 14, it

is seen that the latter of the two shows more mismatches between the solution tuple sizes. This seems to be the affect of choosing input data sets containing words of higher length. This affect is seen more clearly in the graph of Figure 15. In this graph the x-axis represents input data sets of increasing word length. The y-axis represents the average of the solution tuple sizes obtained over 40 separate runs of each data set. Towards the higher word lengths, the mismatches between the solution tuple sizes seem to occur more frequently. Thus the quality of the solutions obtained by the genetic algorithm worsens as the word length increases. Considering the efficiency of the genetic algorithm on data sets of higher word lengths, as demonstrated in the previous section, this loss of solution quality is acceptable. For the most part, the solutions obtained by the genetic

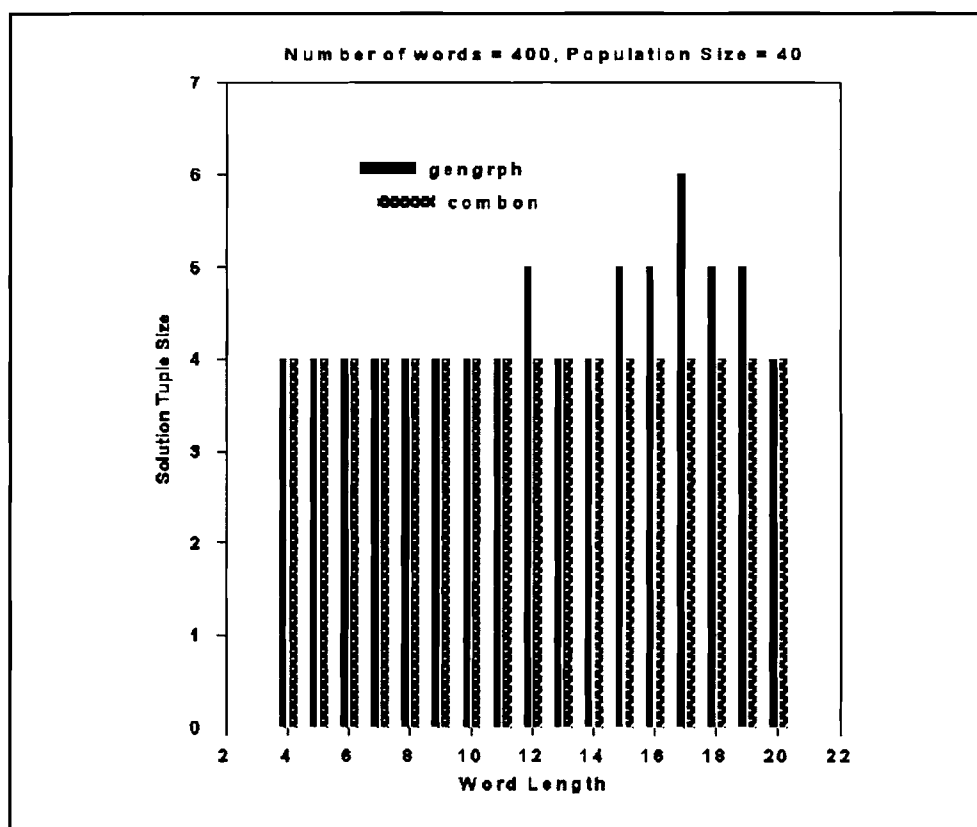


Figure 15: Solution quality for words of increasing length

algorithm, when not optimal, differ in length from the optimal by utmost 3 characters.

In the graph of Figure 14, the x-axis has 19 points. As mentioned before, the average solution tuple size over 40 independent data sets was computed for each of these points on the x-axis. This gives a total of 19×40 or 760 runs for this graph alone. Similarly the graph of Figure 13 represents the outcome of another 760 runs. Similar runs were performed for two additional graphs of this type, for which the word lengths were fixed at 8 and 12. Thus the grand total number of runs performed was 4×760 or 3040. A shell program, designed to perform this simulation, keeps track of the number of times the genetic algorithm yielded a solution of fitness value less than 1.0. These are the times that the genetic algorithm failed to find a solution. For this simulation set this happened 4 times, yielding an almost perfect success rate of $3036/3040$ or 99.86 %.

4.4 Biased Initialization Vs Random Initialization

Two methods of creating the initial population are implemented, namely random initialization and biased initialization. This was discussed in section 3.1.3 of chapter 3. The effect of using biased initialization on the performance in contrast to the random initialization is observed.

In the graph shown in Figure 16, the x-axis represents the data sets with varying word lengths and the y-axis represents the execution time in microseconds. The execution time was computed as an average over 40 separate runs using the corresponding data set on the x-axis. The word length of the data sets increase along the

x-axis, but the number of words in all the data sets is set to 400. Figure 17, shows the quality of the solutions obtained by these runs.

From Figure 16, we see that the performance of both the implementations, as far as the execution time is concerned, does not differ much. Just as the gengrph curve shown in Figures 11 and 12 , the curves in this graph exhibit the same shape and form.

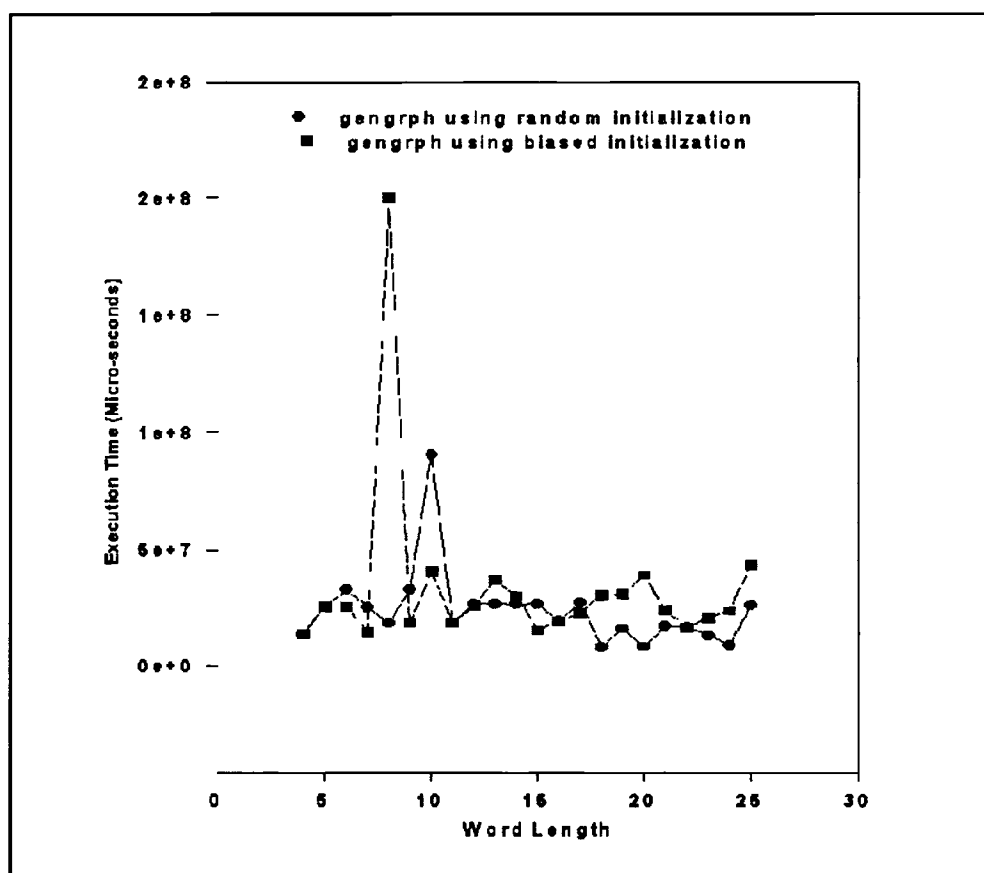


Figure 16: Comparison of performance of two initialization techniques

The increase in word length does not seem to effect the execution time, which stays almost level.

The real differences in the performance between these two implementations of the genetic algorithm is seen in the next graph in Figure 17, which shows the solution tuple sizes obtained. The x-axis is same as that of the graph in Figure 16. The y-axis represents the size of the solution tuples. These sizes were computed as an average over 40 separate runs using the corresponding data sets on the x-axis. The solutions obtained by the implementation that uses biased initialization seem to be better than the ones obtained by the other implementation. This is due to the fact that the solution tuple sizes

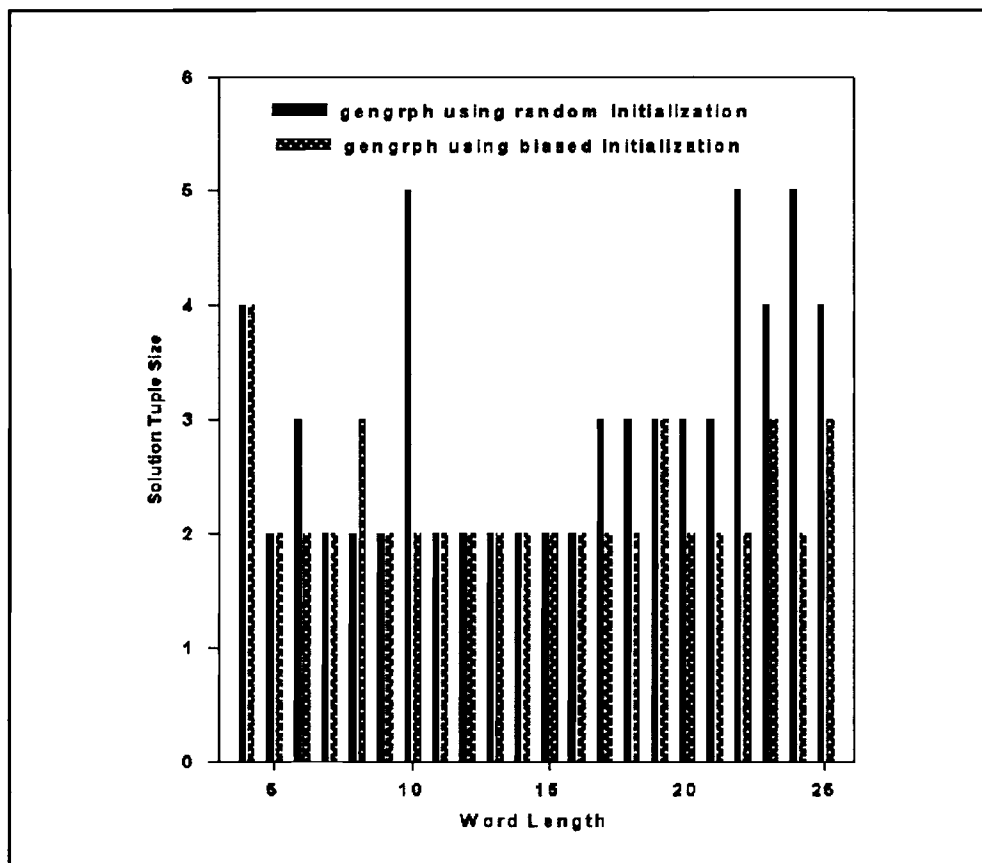


Figure 17: Solution quality corresponding to the graph of Figure 16

are smaller for the genetic algorithm that uses biased initialization even when the fitness of the solutions for both the implementations are equal.

Consider for example the data set corresponding to the word length of 10. The graph in Figure 17 shows that the solution obtained by the implementation that uses the biased initialization is much better, even though, as seen in Figure 16, it takes more execution time than the implementation that uses random initialization. Now consider the data set corresponding to the word length of 9. From Figure 16, it is seen that the implementation using random initialization takes more execution time than the other implementation. However, the solution tuple size obtained by both are the same. So it appears that even though the genetic algorithm that uses biased initialization occasionally takes longer to find a solution, it finds a better solution. This cannot be said for the genetic algorithm that uses random initialization.

4.5 Comparison of the two implementations of the genetic algorithm

The libGA implementation of the genetic algorithm is called *tuple-app* and the C programming language implementation is called *genrph*. The convergence to the solution by both these implementations is expressed in a graph form. The graph represents the standard deviation and average of the fitness of all the chromosomes in each generation. The point where the standard deviation curve reaches the zero level represents the point of convergence. At this point the average curve represents the fitness of the solution to which the genetic algorithm converged.

The graph in Figure 18 shows the convergence to a solution by both the implementations. In this graph, both implementations do not use elitism. It is seen that both the implementations seem to converge at the same rate. The graph in Figure 19 shows the same phenomenon, but both the implementations now use elitism. Although both the implementations still converge at the same rate, a marked improvement is noted from the graphs shown in Figure 18.

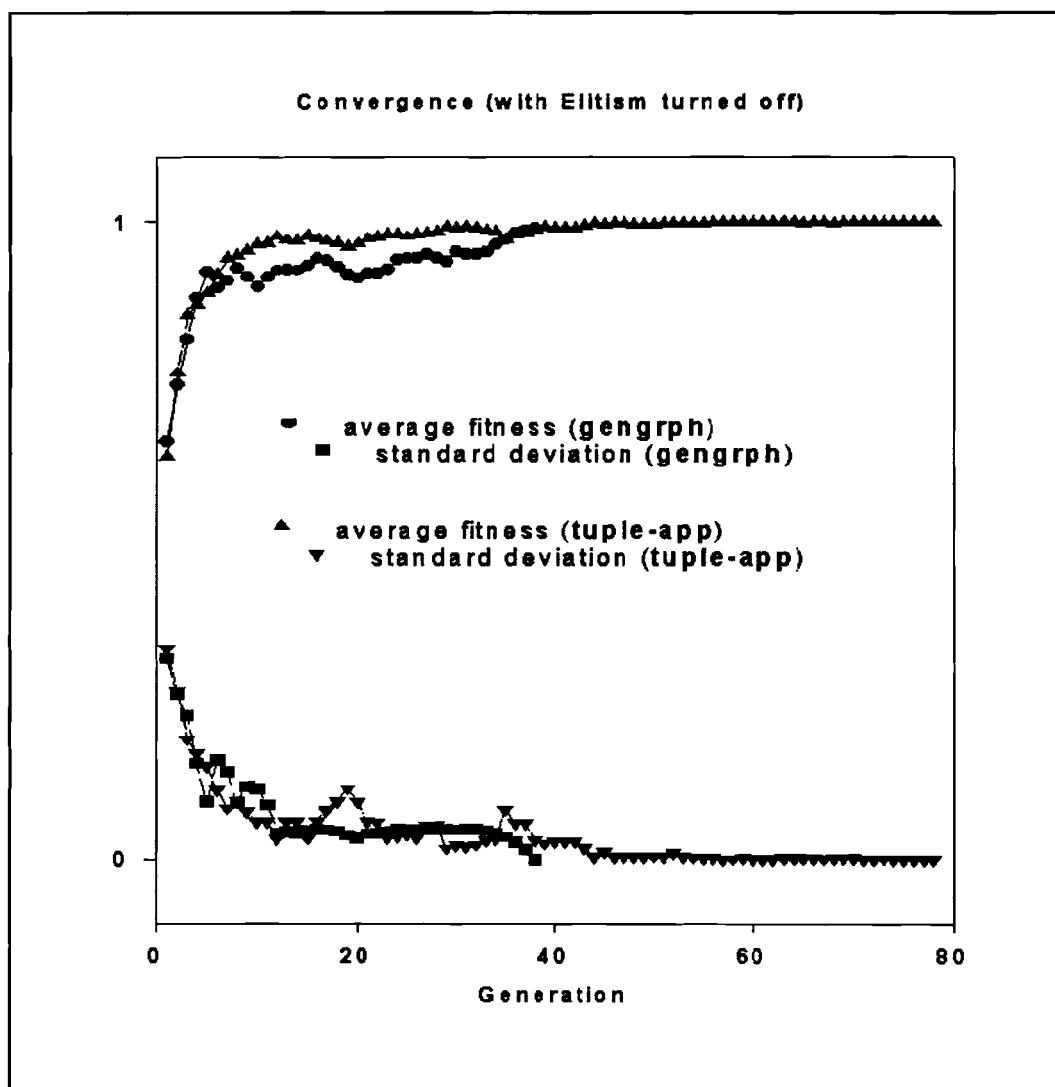


Figure 18: Convergence when not using elitism

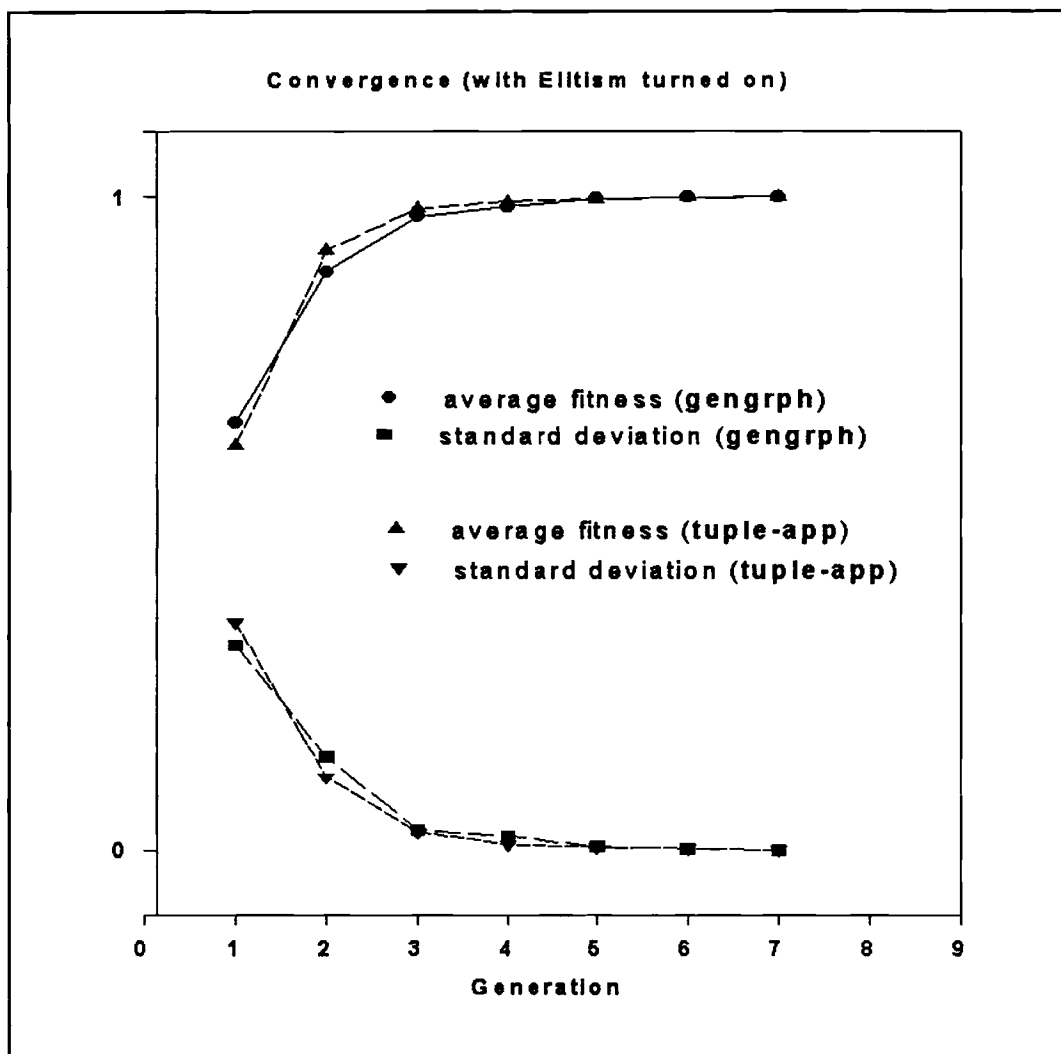


Figure 19: Convergence when elitism is used

4.6. Additional graphs presented in Appendix A.

In section 4.2, it was mentioned that the performance graphs use two types of x-axes: one in which the word length is fixed and the number of words varied, the other in which the number of words is fixed and the word length varied. The graphs in

Figures 8, 9, and 10 represent the first kind. A computation of the success rate of the genetic algorithm for these 3 graphs was given in section 4.3. Appendix A shows graphs that represent the solution quality for the performance graphs shown in Figures 11 and 12. The computation of the success rate of the genetic algorithm for these set of graphs is also given in the appendix A. In section 4.4, a comparison of the performance of the two initialization techniques was presented. The graph in Figure 16 compared the execution times and the graph in Figure 17 compared the solution qualities. Appendix A shows more graphs of these type.

CHAPTER 5

5 CONCLUSIONS AND FUTURE WORK

5.1 Conclusions

A genetic algorithm was implemented to solve the problem of extracting tuples from word sets. Two implementations were realized, one using the C programming language, and the other using the libGA application tool. The performance of these two implementations were similar. The implementation of genetic algorithms is easier using the libGA application tool, since only the fitness function needs to be coded. To implement more complex genetic program, the libGA application tool does not provide the flexibility needed to alter the data structures and provide decoders and repair algorithms. The libGA application tool can handle implementations of only classical genetic algorithms. The C programming implementation of the genetic algorithm used two separate techniques of initializing the population. The first used a random initialization that involves a random selection of points from the entire solution domain. The second used a biased initialization that involves selection of points based on estimated probabilities of overlap for each separate tuple size. When using the random initialization technique the genetic algorithm converged to a solution in less time, however the biased version seemed to converge to a better solution. The execution time of the genetic algorithm to converge to a solution increased with the population size used.

But using larger population sizes yielded better solutions. The performance of the genetic search was compared to an exhaustive search.

The genetic search seemed to perform better under certain conditions and the exhaustive search under other conditions. For smaller word lengths the exhaustive search outperformed the genetic search. As words of larger length were considered the performance of the genetic search improved until finally it outperformed the exhaustive search. The solution domain increases exponentially with the word length. The genetic search seemed to perform better for larger solution domain. This performance was not gained without paying a price. The price paid was the length of the solution tuple. The solution tuple lengths were larger than the optimal solution tuple lengths for larger word lengths, but this was under tolerable limits. The gain in the performance of the genetic algorithm outweighed the loss in the optimality of the solution tuples. For word lengths of 20 and more the length of the solution tuple was larger than the optimal length by utmost 3 or 4 characters. Although the exhaustive search guarantees an optimal solution, this search is combinatorially explosive. The genetic algorithm on the other hand shows an approximate linear increase in the search time and finds a solution with 85% chance of being optimal. Even when word length was fixed and the input word list was increased, the genetic search eventually outperformed the exhaustive search.

5.2 Future Work

The population size effects the performance. Larger populations yield better

solutions but also take longer time to converge. Choosing an appropriate population size for this problem is an issue that can be addressed in future. Also the affect of using different variations in the implementations of the genetic operators on the performance can be explored. The evaluation scheme can be modified to incorporate the probabilities that were computed in section 3.1.3.2 of Chapter 3. A suggestion would be to use these probabilities to form a biased fitness evaluation scheme.

BIBLIOGRAPHY

- [ARU93] S. Arunkumar and T. Chockalingam, "Genetic Search Algorithms and their randomized Operators", Computers and Mathematics with Applications, Vol 25, (5), (March 1993), pp 91-100
- [AOE92] Jun-Ichi Aoe, Katsushi Morimoto and Takashi Sato, "An Efficient Implementation of Trie Structures", Software-Practice and Experience, Vol 22, (9), (April 1992), pp 695-721.
- [BAT93] David L. Battle and Michael D. Vose, "Isomorphisms of genetic algorithms", Artificial Intelligence, Vol 60, (March 1993), pp 155-165
- [BOR91] Jurgen Borstler, Ulrich Moncke, and Reinhard Wilhelm, "Table Compression for Tree Automata", ACM Transactions on Programming Languages and Systems, Vol 13, (3), (July 1991), pp 295-314.
- [BRA89] Marshall D. Brain and Alan. L. Tharp, "Near-perfect hashing of large word sets", Software-Practice and Experience, Vol 19, (10), (October 1989), pp 967-978.
- [CEL93] Joe Celko, "Genetic Algorithms and Database Indexing", Dr. Dobb's Journal, Vol 18,(4), (April 1993), pp 30-34
- [CHA91] C. C. Chang, C. Y. Chen and J. K. Jan, "On the Design of a Machine-Independent Perfect Hashing Scheme", The Computer Journal, Vol 34 (5), (1991), pp 469-474.
- [CHI91] Chin-Chen Chang and Tzong-Chen Wu, "A Letter-oriented Perfect Hashing Scheme Based upon Sparse Table Compression", Software-Practice and Experience, Vol 21, (1), (January 1991), pp 35-49.
- [CHA86] C. C. Chang and R. C. T. Lee, "A letter-oriented Minimal Perfect Hashing Scheme", The Computer Journal, Vol 29, (3), (1986), pp 277-281
- [CHA84] C. C. Chang, "The study of an ordered minimal perfect hashing scheme", Communications of the ACM, Vol. 27, (4), (1984), pp 384-387.
- [CIC80] Richard J. Cichelli, "Minimal Perfect Hash Functions Made Simple", Communications of the ACM, Vol. 23, (1), (January 1980), pp 17-19.
- [DAM91] Eric S. Lander, Robert Landridge and Damian M. Saccocio, "A report on

computing in molecular biology; mapping and interpreting biological information", Communications of the ACM, Vol 34, (November 1991) pp 32-39.

- [GOL89] David E. Goldberg, "Genetic Algorithms in Search, Optimization, and Machine Learning", Reading, Mass: Addison-Wesley.
- [GOL85] David E. Goldberg, "Genetic Algorithm and Rule Learning in Dynamic Control Systems", Proceedings of the First International Conference on Genetic Algorithms, Lawrence Erlbaum Associates, Hillsdale, NJ, (1985), pp. 8-15.
- [HOL92] John H. Holland, "Genetic Algorithms", Scientific American, Vol 267,(1), (July 1992), pp 66-72.
- [HOF85] Douglas R. Hofstadter, "The genetic code: arbitrary?", in Metamagical Themas, New York: Basic Books, pp 671-699.
- [HOL75] John Holland, "Adaptation in Natural and Artificial Systems", University of Michigan Press, 1975.
- [HOL73] John H. Holland, "Genetic algorithms and the optimal allocation of trials", SIAM J. Computing, Vol 2, (2), (June 1973), pp 50-56
- [HUN91] Lawrence Hunter, "Artificial Intelligence and Molecular Biology", AI Magazine, Vol 11, (January 1991), pp 27-36.
- [JAS80] G. Jasechke and G. Osterburg, "On Chichelli's Minimal Perfect Hash Functions Method", Communications of the ACM, Vol. 23, (12), (December 1980), pp 728-729.
- [KER70] B. W. Kernighan and S. Lin, "An Efficient heuristic procedure for partitioning graphs", Bell Systems Technical Journal, Vol 49, (2), (1970), pp 291-308
- [LAN91] Eric S. Lander, Robert Landridge and Damian M. Saccocio, "Computing in molecular biology: mapping and interpreting biological information", Computer, Vol 24, (November 1991), pp 6-13
- [LAD71] R. Landauer and J. W. Woo, "Minimal Energy Dissipation and Maximal Error for the Computational Process", J. Appl. Phys., Vol 42, (1971), pp 2301+.
- [LIN70] Shen Lin, "Computer Solutions of the Traveling Salesman Problem", Bell

Systems Technical Journal, Vol , (December 1965), pp 2245-2265.

- [LIU91] Joseph W. H. Liu, "A Generalized Envelope Method for Sparse Factorization by Rows", ACM Transactions on Mathematical Software, Vol 17, (1), (March 1991), pp 112-129.
- [MIC92] Zbigniew Michalewicz, "Genetic Algorithms + Data Structures = Evolution Programs", Springer-Verlag Berlin Heidelberg, (January 1992)
- [PAV92] Pevzner A. Pavel, "Multiple alignment, communication cost, and graph matching", SIAM Journal on Applied Mathematics, Vol. 52, (December 1992), pp 1763-1779.
- [SPI92] Richard Spillman, "Genetic Algorithms", Dr. Dobb's Journal, Vol 18,(4), (July 1992), pp 26-30
- [SAG85] Thomas J. Sager, "A Polynomial Time Generator for Minimal Perfect Hash Functions", Communications of the ACM, Vol. 28, (5), (May 1985), pp 523-532.
- [TAR79] Robert Endre Tarjan and Andrew Chi-CHih Yao, "Storing a Sparse Table", Communications of the ACM, Vol. 22, (11), (November 1979), pp 606-611.
- [VOS91] Michael D. Vose, "Generalizing the notion of schema in genetic algorithms", Artificial Intelligence, Vol 50, (1991), pp 385-396
- [WAL89] Walbridge, Charles T., "Genetic Algorithms : What Computers Can Learn from Darwin", Technology Review, Vol 92 (1), (Jan 1989), pp 46-53

APPENDIX A

The graph in Figure 20 shows the comparison of execution times required to find a solution by the genetic search and exhaustive search. Word length varies along the x-axis. The number of words in each data set is fixed at 800. This graph is an addition to the graphs of figures 11 and 12, in which the size of the data sets were fixed at 400 and 600 words respectively. The graph on the next page shows the quality of the solutions obtained for the same data sets as depicted in Figure 20.

The graphs in figures 23 through 26 compare the performance between the two implementations of the genetic algorithm which use random and biased initialization respectively. Graphs in figures 23 and 25 compare the execution times and the graphs in figures 24 and 26 compare the solution qualities.

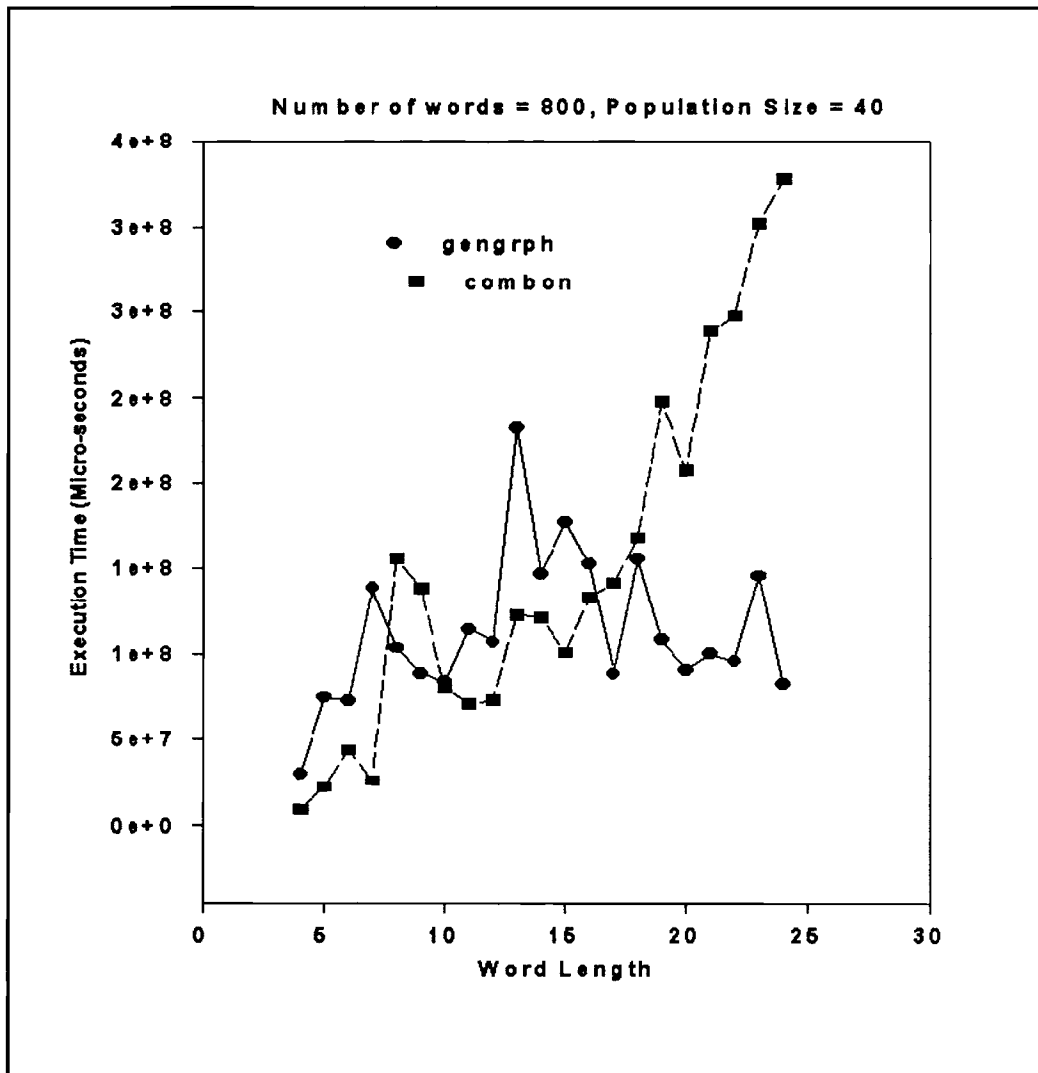


Figure 20: Performance graph #3 for words of increasing length

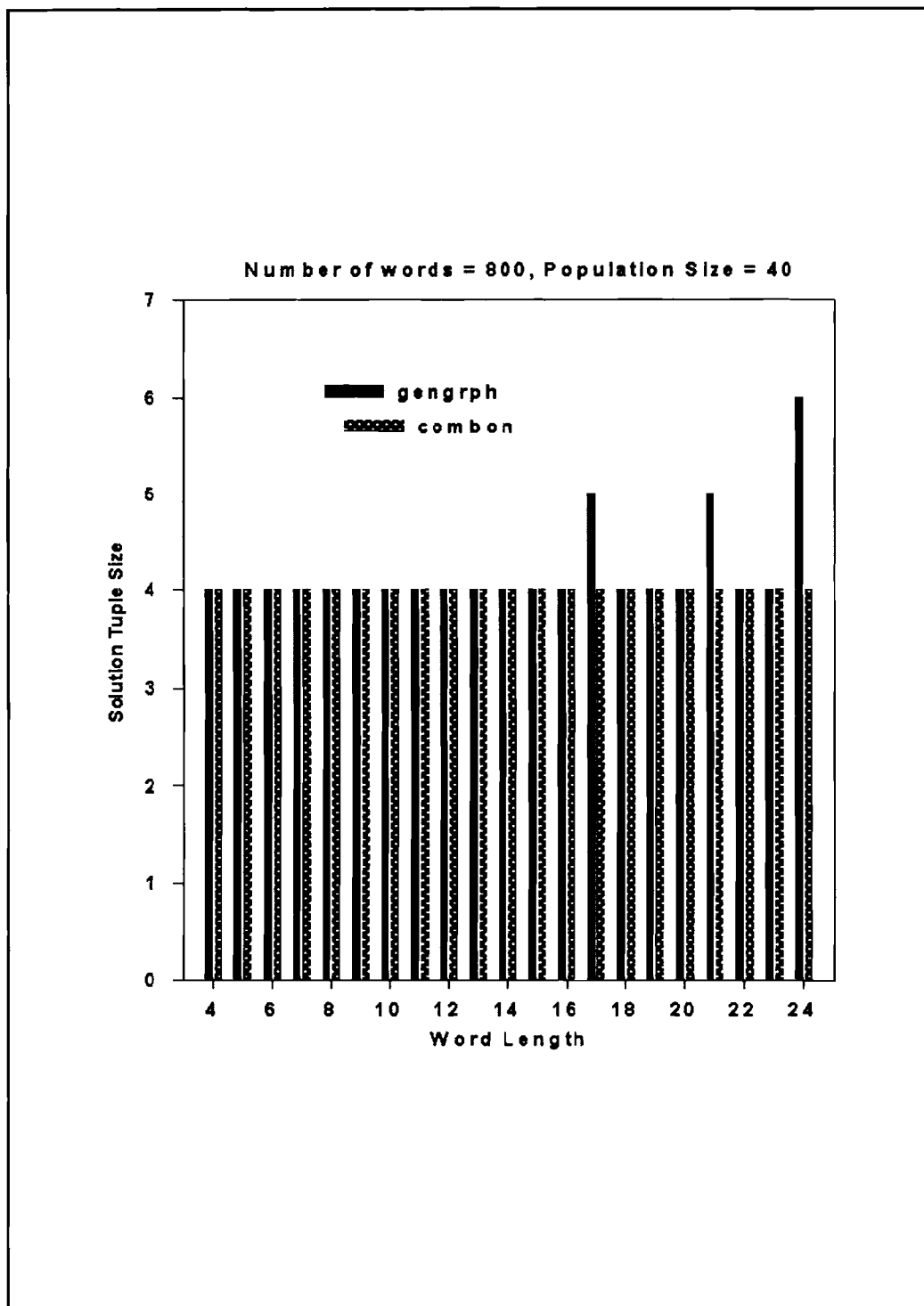


Figure 21: Solutions corresponding to the graph in figure 20

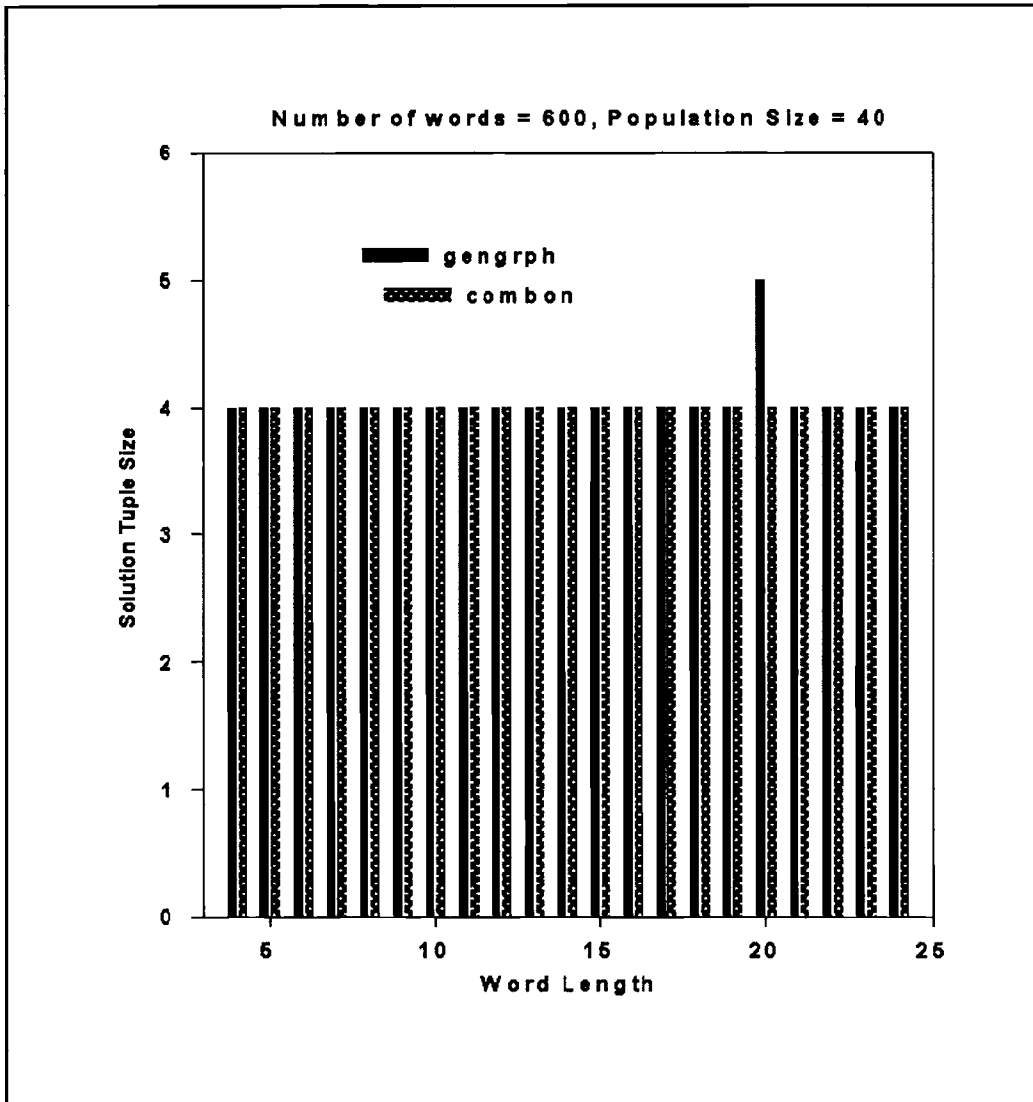


Figure 22: Solutions corresponding to graph in figure 12

The graphs shown in figures 11, 12, and 20 belong to the category in which the length of the words in the input data sets increase along the x-axis. The only difference between these graphs is the size of the input data set. For the graph in figure 11 all the data sets have 400 words. Similarly for the graph in figure 12 the size of input data sets are set at 600 words, and for figure 20 the size is set at 800 words. Each of these graphs are constructed from 24×40 independent runs, because each point on the y axis is computed as an average over 40 runs and there altogether 24 points. Thus the total number of runs performed in all the three graphs is $3 \times 24 \times 40 = 2880$. The number of times the genetic algorithm failed to find a solution of fitness 1.00 is 57, which gives a success rate of $(2880 - 57) / 2880 = 98.02\%$.

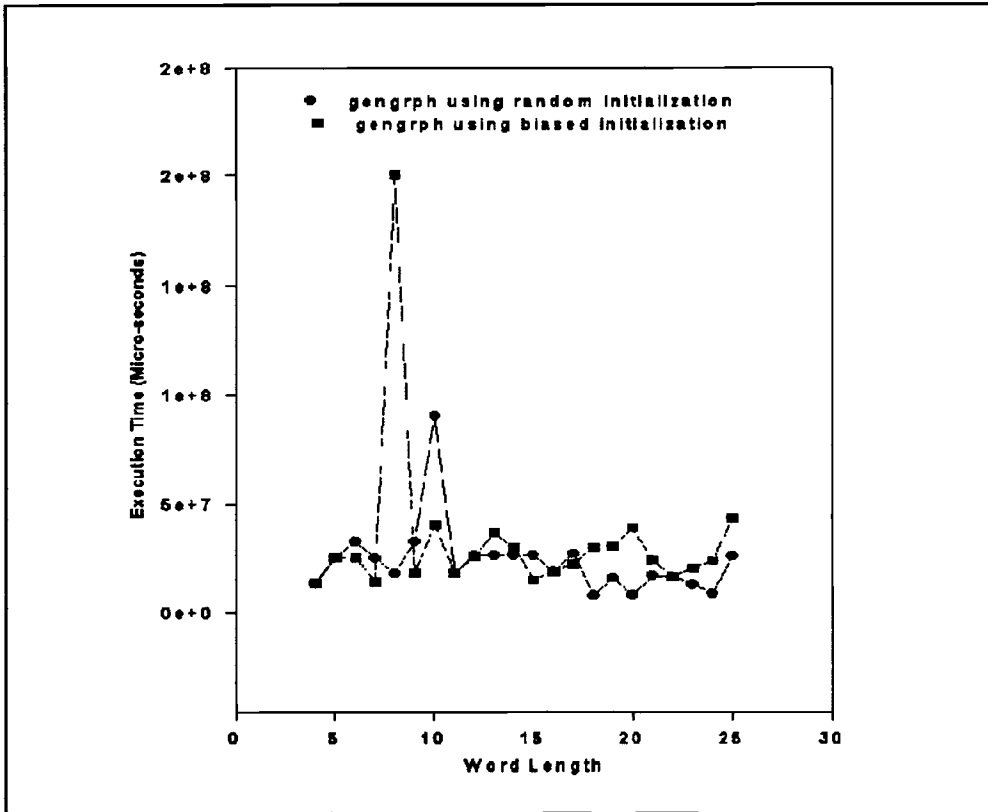


Figure 23: Comparison of performance of two initialization techniques.
All input data sets contain 400 words

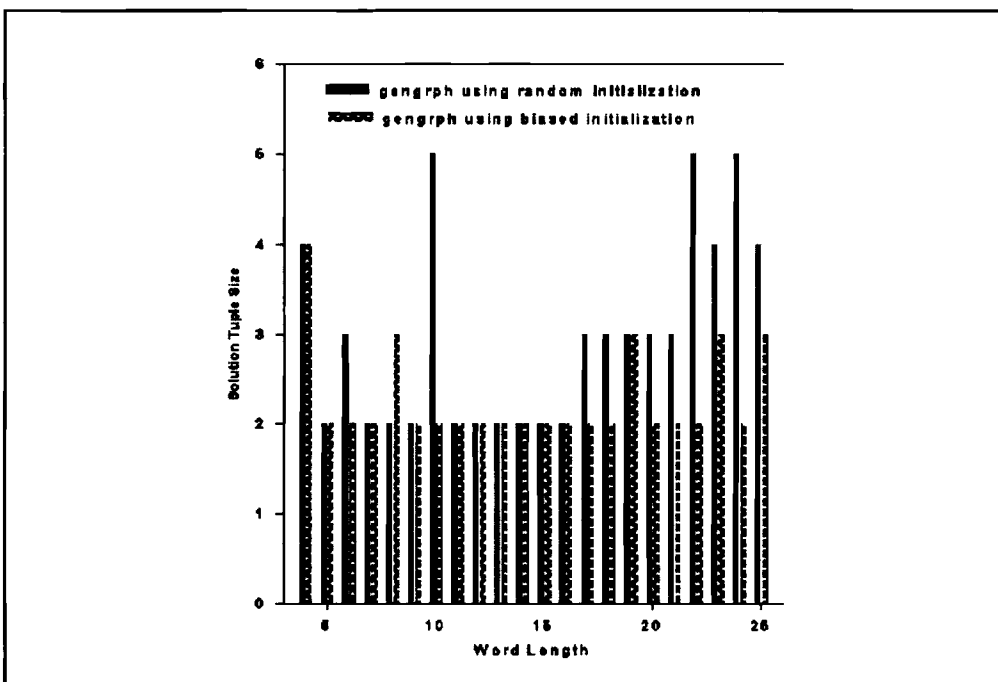


Figure 24: Solution quality corresponding to the graph of Figure 23

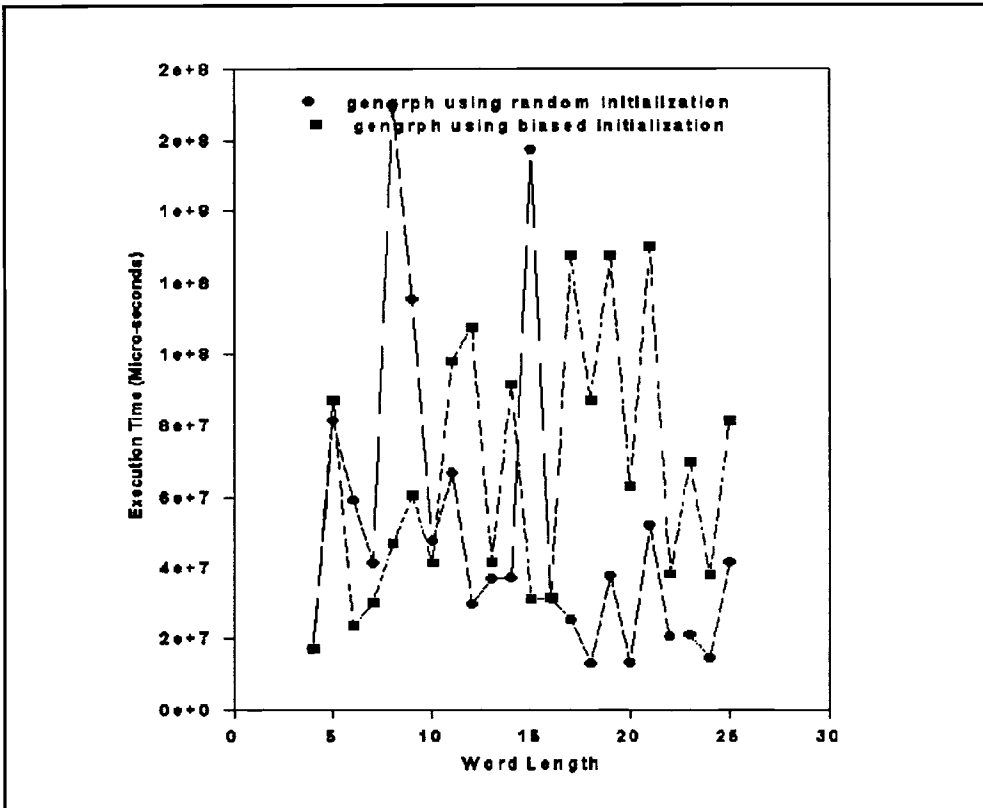


Figure 25: Comparison of performance of two initialization techniques. All input data sets contain 600 words

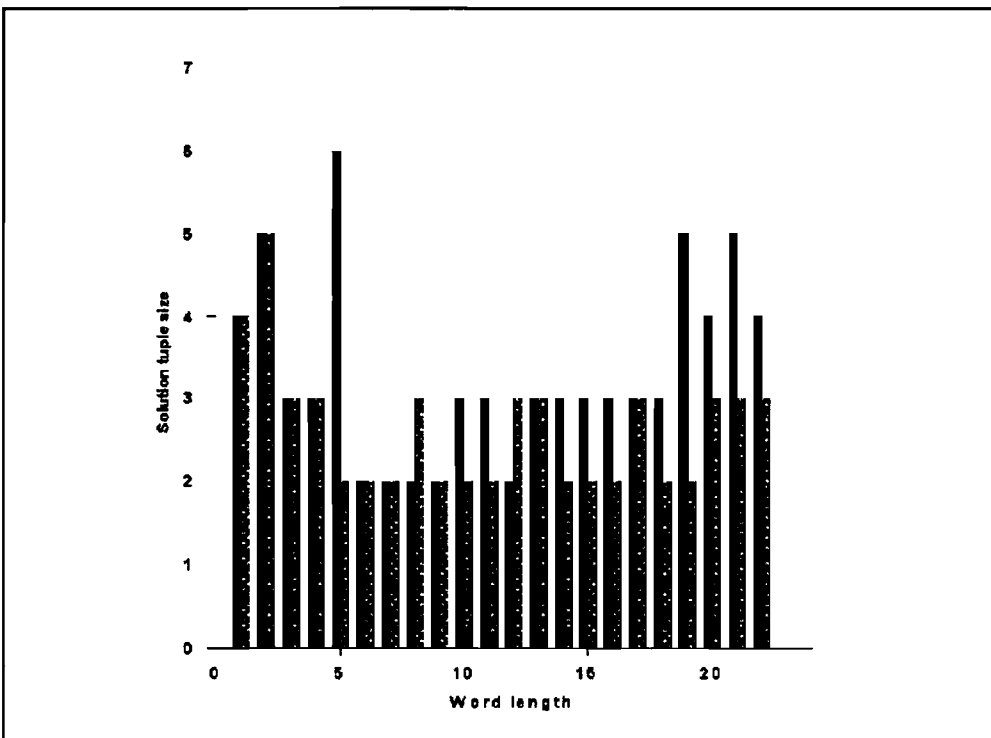


Figure 26: Solution quality corresponding to the graph of figure 25

APPENDIX B

The source code of the genetic algorithm *gengrph* implemented in C

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <math.h>

#include "const.h"    /*Constants used */
#include "def.h"      /*Data Structures defined*/

#include "rand.h"     /*Random number generation related functions*/
#include "print.h"    /*Output functions*/
#include "eval.h"     /*Functions related to fitness evaluation */
#include "genop.h"    /*Functions related to genetic operations */

/*****
Driver Module of the Genetic Algorithm.

Creates a new generation from an existing one. The
population continues to evolve through the generations
until a solution is found, or the population is dominated
by a single chromosome/individual.
*****/
main (argc,argv)
int argc;
char *argv[];
{
Population    pop, npop;    /* old and new populations */

int           maxndx,      /* index to the chromosome of maximum fitness*/
             minndx,      /* index to the chromosome of minimum fitness*/
             gencnt;      /* Generation counter */

float         avg, av, stdev; /* average and standardeviation */

int           i, j,
             lchrom,      /* length of chromosome */
             nofwords,    /* total number of words in the input file*/
             popsize,     /* population size */
             stop, stpndx; /* stop is used to store a boolean 1 or 0
                           for termination condion. */
```

```
char    letters[ORDINAL], /* The alphabet used by the input word set */
        words[MAXNUMBEROFTUPLES][MXLNGTH]; /* the list of
                                           input words */
```

```
float   scale [MAXPOP];      /*cumulative sumfitness used in selection
                             process*/
```

```
/* The command line syntax : consists of the file name that contains the
   input data, the population size of a generation, and a boolean value for elitism
   of 1 or 0. If elitism is 1 then elitism is applied otherwise no elitism is
   applied*/
```

```
if (argc !=4 ) {
    printf ("\nFormat : %s <input file> <pop size> <elitism>\n", argv[0]);

    exit (1);
}
```

```
clock (); /* Clock is reset to 0. The next time clock is invoked upon termination
           of the genetic algorithm to determine the amount of cpu time used to
           execute the program. */
```

```
MXPOP = atoi (argv[2]);
ELIT = atoi (argv[3]);
```

```
/* This module generates the initial pool of chromosomes that the first
   generation will consist of. The initial pool is created using either a random
   initialization or a biased initialization.*/
```

```
Initialize (pop, &popsiz, scale, argv[1], &lchrom,
           letters, words, &nofwords);
```

```
gencnt = 1; /* Start the generation counter */
do {
```

```
    /* This module creates a new population by applying the genetic operators:
       reproduction, crossover, and mutation to selected chromosomes
       of the old population */
```

```
    Generation (pop, npop, &popsiz, lchrom, scale,
               words, nofwords);
```

```

/* The old population is replaced by the new population */
for (i=0;i<popsiz;i++) pop[i] = npop[i];

/* Increment the generation counter to indicate the creation of the next
generation*/
gencnt++;

/* Compute relevant statistics for the new generation*/
statistics (popsiz, &maxndx, &avg, &minndx, pop);

/* This module tests for the termination of the Genetic Cycle. If
the module returns with a stop value of 1 then the cycle/loop
terminates */
Terminate (pop, popsiz, &stop, maxndx);

} while (!stop);

printf ("\n%s", pop[maxndx].chrom); /* The genetic program converged to this
solution */
}

```

file: const.h (constants used)

```

#define MAXPOP      500          /*Maximum population size*/
#define ORDINAL     26          /* Number of unique alphabets contained
by the list of words */
#define MXLNGTH     31          /* Maximum String Length of all the
words */
#define MAXNUMBEROFTUPLES 1001 /*Maximum number of tuples*/

```

file: def.h (Data structures used)

```

typedef int BOOL;
typedef char Allele;
typedef Allele Chromosome[MXLNGTH];
typedef struct {
    Chromosome   chrom;
    float        x;
    float        fitness;
    int          parent1, parent2, xsite;
    int          count;
} Individual;

```

```

typedef Individual Population[MAXPOP];

int nmutation, ncross, MXPOP = 0, ELIT=0;
float pcross, pmutation, sumfitness; /* probability of crossover, mutation. */

long RestrainLength [MXLNGTH+1]; /* used in the fitness function to
update the smallest hashlength for each
tuple size */

```

file: rand.h

```

long seed = 1.0;

/*****
Function : Random ()
Fetch a single random number between 0.0 and 1.0
*****/
float Random()
{
float random;

long temp_seed; /* temp_seed is declared as long ie, 32 bit accuracy */
int aa=16807; /* { 7^5 } */
long m=2147483647; /* {2 ^31 -1 } */
int q=127773; /* {m / aa}*/
int r =2836; /* {m mod a} */

temp_seed=aa*(seed % q) - r* ((int)(seed/q));

if (temp_seed >= 0) /* if the temp_seed is greater than
zero seed is made equal to temp_seed
otherwise m is added to temp_seed to
make it positive. */

seed = temp_seed;
else
seed =(temp_seed + m);

random =(float)seed/m; /* seed is divided by m to get the
random number . To preserve the
accuracy of random the float value
is taken */

return(random); /* returning the Random number */

```

```

}

/*****
Function : Rnd (low .. high)
Returns an integer selected uniformly and pseudorandomly
between upper and lower limits.
*****/
int Rnd (low, high)
int low, high;
{
int i;
if (low >= high)
    i = low;
else
    {
    i = (Random() * (high-low+1) + low);
    if (i > high)
        i = high;
    }
return i;
}

```

file: genop.h

```

/*****
The Initialization Module

This function is responsible for the initialization of all
the parameters, and data structures needed for use by
other modules. Among various other initializations, the
gene pool for the first generation is initialized here.
The input data, the word list, is read from the input file.
*****/
Initialize (pop, popsize, scale, filename, lchrom, letters,
           words, nw)
Population    pop;
int           *popsize;
float        scale[];
char         *filename;
int          *lchrom;
char         letters[];
char         words[][MXLNGTH];
int          *nw;
{

```



```

int      i, end, noletters, nofindivid, n, tuplesize, pndx;
char     msg [MXLNGTH], ch;
float    objfunc (), sum, power (), find_series_sum ();
FILE     *fp;

/* Initialize the probabilities of application of the genetic operators*/

pmutation = 0.001;    /* Probability of mutation/mutation rate */
nmutation = 0;        /* Counter to keep track of the number of
                       mutations performed */

pcross    = 0.25;     /* Probability of crossover/crossover rate */
ncross    = 0;        /* Counter to keep track of the number of
                       crossover operations performed */

/* Opening the input file that contains the list of words */
if ((fp = fopen(filename, "r")) == NULL) {
    printf ("\nError in opening the input file containing the words.\n");
    exit (1);
}

/* The data structure : RestrainLength, is used in the eval.h header file
   to keep check on the size of the tuples. */
for (i = 1; i <= MXLNGTH; i++)
    RestrainLength[i] = 80000L;

/* Read in the header section of the input file, which contains the
   the parameters : nw (number of words ), lchrom (word length),
   and the letters (the alphabet comprising the words) */
fscanf (fp, "%d %d ", nw, lchrom);
i = 0;
while ((ch = getc(fp)) != '\n') letters[i++] = ch;
noletters = i;
letters[i] = '\0';

/* Read in the words from the input file*/
i = 0;
while (fscanf (fp, "%s", words[i]) != EOF) i++;
fclose (fp);

/* Computing the size of the complete solution domain. If the
   lchrom is 5 then the population will consist of chromosomes
   ranging from : 00001 to 11111, which totals to 64 - 1 = 63 */
for (end=0,i=0;i < *lchrom;i++)

```

```
end = (end << 1) | 0x01;
```

```
/* if size of solution space is greater than the maximum population
size then population size is set at MXPOP, otherwise population
size is set at the size of solution domain.
```

```
Also if size of solution domain > MXPOP then the initial population
will consist of MXPOP randomly selected entries from entire solution
domain, otherwise the initial population will be the entire solution
domain. */
```

```
if (end > MXPOP){
  if (INITYPE == 0){ /*Implementing Random Initialization */
    *popsize = MXPOP;
    for (i = 0; i < *popsize; i++) {
      convertobin (Rnd(1,end), msg, *lchrom);
      strcpy (pop[i].chrom, msg);
    }
  }
  else {
```

```
/* Implementing Biased initialization based on probability of overlap
among tuples calculated as explained in section 3.1.3.2.
From section 3.1.3.2 it was stated that the probability that two k-tuples
will not be identical is  $(r^n - r^{n-k}) / (r^n - 1)$ . The number of k-tuples that will
be introduced into the population will be biased by the weight given by
```

$$weight = \frac{r^n - r^{n-k}}{\sum_{k=1}^n (r^n - r^{n-k})} = \frac{1 - 1/r^k}{\sum_{k=1}^n (1 - 1/r^k)}$$

```
So the number of k-tuples = weight x total number of words*/
*popsize = MXPOP;
pndx = 0;
nofindivid = 0;
```

```
/* This function computes the denominator of the expression for weight
given above*/
sum = find_series_sum (nofletters, *lchrom);
for (tuplesize=1; tuplesize <= *lchrom; tuplesize++) {
  /* the number of tuples of size tuplesize that will be introduced into the
population is computed */
  n = *popsize * ((1 - 1/power(nofletters, tuplesize))/sum);
  /* these tuples are now introduced into the population */
```

```

        for (i = 0; i < n; i++) {
            createAtupleofsize (tuplesize, msg, *lchrom);
            strcpy (pop[pndx++].chrom, msg);
        }
    }
    if (pndx < *popsize-1)
        for (i=pndx; i < *popsize; i++) {
            createAtupleofsize (*lchrom, msg, *lchrom);
            strcpy (pop[i].chrom, msg);
        }
    }
else {
    *popsize = end;
    for (i=0; i < end; i++) {
        convertobin (i+1, msg, *lchrom);
        strcpy (pop[i].chrom, msg);
    }
}

/* Initialize the sum total of the fitness of all the chromosomes */

for (sumfitness = 0, i = 0; i < *popsize; i++) {
    pop[i].fitness = objfunc (pop[i].chrom, *lchrom,
        words, *nw);
    sumfitness += pop[i].fitness;
}

/* Initialize the cumulative sum of the fitness of all the chromosomes */

scale[0] = pop[0].fitness;
for (i=1; i < *popsize; i++)
    scale[i] = scale[i-1]+pop[i].fitness;
}

```

/******

Generation ()

Create a new generation through select, crossover,
and mutation.

Generation size remains same when an even numbered popsize
is considered. If the popsize is odd then the resulting
new generation will contain one extra chromosome.

```

*****/
Generation (popold, popnew, popsize, lchrom, scale, words, nw)
Population  popold, popnew;
int        *popsize, lchrom;
float      scale[];
char       words[][MXLNGTH];
int        nw;
{
int j, mate1, mate2, jcross, i;
Allele Mutation ();
float objfunc();

j = 0;

do {
    mate1 = Select (*popsize, sumfitness, popold, scale);
    mate2 = Select (*popsize, sumfitness, popold, scale);
    Crossover (popold[mate1].chrom, popold[mate2].chrom,
              popnew[j].chrom, popnew[j+1].chrom, lchrom);

    popnew[j].fitness = objfunc (popnew[j].chrom, lchrom, words, nw);
    popnew[j+1].fitness =
        objfunc (popnew[j+1].chrom, lchrom, words, nw);

    /* Replace worst child with parent1 if parent1 is better */
    if (popnew[j].fitness < popnew[j+1].fitness) {
        if (popold[mate1].fitness > popnew[j].fitness)
            popnew[j] = popold[mate1];
    }
    else {
        if (popold[mate1].fitness > popnew[j+1].fitness)
            popnew[j+1] = popold[mate1];
    }

    /* Replace worst child with parent2 if parent2 is better */
    if (popnew[j].fitness < popnew[j+1].fitness) {
        if (popold[mate2].fitness > popnew[j].fitness)
            popnew[j] = popold[mate2];
    }
    else {
        if (popold[mate2].fitness > popnew[j+1].fitness)
            popnew[j+1] = popold[mate2];
    }
}

```

```

    j += 2;

    } while (j < *popsize);

*popsize = j;

scale[0] = popnew[0].fitness;
for (i=1;i < *popsize;i++)
    scale[i] = scale[i-1]+popnew[i].fitness;

}

/*****
Function : Mutation ()
*****/
Allele Mutation (alleleval)
Allele alleleval;
{
int mutate;
mutate = Flip (pmutation);
if (mutate) {
    if (alleleval = '0') return '1';
    else return '0';
}
else
    return (alleleval);
}

/*****
Crossover ()
Cross 2 parent strings, place in 2 child strings
*****/
Crossover (parent1, parent2, child1, child2, lchrom)
Chromosome parent1, parent2, child1, child2;
int lchrom;
{
int jcross;
int pcnt, ccnt=0;
char tmpstore[MXLNGTH];
Allele Mutation ();

if (Flip (pcross)) {
    jcross = Rnd(1, lchrom-1);
}

```

```

else {
    jcross = lchrom;
}

for (pcnt=0;pcnt<jcross;pcnt++) {
    child1[ccnt] = Mutation (parent1[pcnt]);
    child2[ccnt] = Mutation (parent2[pcnt]);
    ccnt++;
}
for (pcnt=jcross;pcnt<lchrom;pcnt++) {
    child2[ccnt] = Mutation (parent1[pcnt]);
    child1[ccnt] = Mutation (parent2[pcnt]);
    ccnt++;
}
child1[ccnt] = child2[ccnt] = '\0';
}

/*****
Function : Select ()
Selects an individual from a Population biased by the fitness
of this individual. (Anologous to Selection on a Roulette Wheel
*****/
int Select (popsize, sumfitness, pop, scale)
int popsize;
float sumfitness;
Population pop;
float scale[];
{
int i;
float Rand;

Rand = Random() * sumfitness;

for (i=0;i<popsize;i++)
    if (Rand < scale[i])
        return i;
return -1; /* error */
}

/*****
Function : Flip (probability)
Returns result of a simulated biased coin toss

```

```

*****/
BOOL Flip (probability)
float probability;
{
if (probability == 1.0)
    return 1;
else
    return (Random() <= probability);
}

/*****
Funtion takes an integer as the input argument,
and converts it to a binary string equivalent.
*****/

convertobin (n,msg,l)
char msg[];
int n,l;
{
int i;

for (i=l-1;i>=0;i--) {
    if (n&1)
        msg[i] = '1';
    else
        msg[i] = '0';
    n=n>>1;
}
}

/*****
Module responsible for determining whether the
termination of the Genetic cyle should be performed
*****/

Terminate (pop, popsize, terminat, maxndx)
Population pop;
int popsize, *terminat, maxndx;
{
int i, count;

*terminat = 0;

count = 0;

/* if population is dominated by a single chromosome type

```

then terminate. i.e When all the chromosomes in the population have identical fitness values, the genetic cycle is terminated*/

```

for (i = 0; i < popsize; i++)
    if (pop[i].fitness == pop[maxndx].fitness)
        count++;

    if (count >= popsize)
        *terminat = 1;
}

/* Computes the maximum, minimum, average, and standard deviation of fitness
values of all the chromosomes in the population */
statistics (popsize, maxndx, avg, minndx, pop)
int popsize, *maxndx, *avg, *minndx;
Population pop;
{
int i;

sumfitness = pop[0].fitness;
*minndx = 0;
*maxndx = 0;

for (i = 1; i < popsize; i++) {
    sumfitness += pop[i].fitness;
    if (pop[i].fitness > pop[*maxndx].fitness) *maxndx = i;
    if (pop[i].fitness < pop[*minndx].fitness) *minndx = i;
}
*avg = sumfitness/popsize;
}

/* Finds the length of the tuple represented by the chromosome. This carried out
by counting the number of 1's in the chromosome. This value is used to plot the
solution quality in terms of tuple length. */
findtuple_length (tuple, lchrom)
Chromosome tuple;
int lchrom;
{
int i, len = 0;
for (i = 0; i < lchrom; i++)
    if (tuple[i] == '1') len++;
return len;
}

```


/* Function is used to compute the probabilities mentioned in section 3.1.3.2. These probabilities are used to implement a biased initialization of the initial population. */

```
float find_series_sum (r, n)
int r, n;
{
int i;
float sum = 0, power ();

for (i=1;i<=n;i++) {
    sum = sum + (1 - 1/power(r,i));
}
return (sum);
}
```

/* Computes the value of a raised to the power of n . This function is also used in the computation of probabilities mentioned in section 3.1.3.2*/

```
float power (a, n)
int a, n;
{
int i;
float prod = 1.0;

for (i = 0; i < n; i++) prod = prod * a;
return prod;
}
```

/******

This function creates a random tuple of size k

*****/

```
createAtupleofsize (k, tuple, l)
```

```
int k;
char *tuple;
int l;
{
int pos[MXLNGTH], j, i, ndx, ps, len;
```

```
len = l;
for (i = 0; i < l; i++) {
    pos[i] = 0;
    tuple[i] = '0';
}
```

```

for (i=0; i < k; i++) {
    ps = Rnd (0, l-1);
    while (pos[ps] == -1) {
        ps++;
        if (ps == len) {
            ps = 0;
        }
    }
    tuple[ps] = '1';
    pos[ps] = -1;
    l--;
}
}

```

file: eval.h

```

/*****
The FITNESS Evaluation Module

```

The objfunc () takes a chromosome as an argument and assigns it a fitness value in the range 0..1. The higher the fitness value the closer the chromosome is to the solution. For fitness value of 1, the chromosome represents the solution.

The chromosome represents a set of tuples. The fitness value gives a measure of the number of duplicate entries existing in the list of tuples. This is computed as an average hashlength = nw/hashlength.

for example : if the chromosome 0101 represents the tuples : aa ab cd aa ba cd ba ba. Then a frequency count of this list will tell us that there are 2 aa's, 1 ab, 2 cd's, and 3 ba's in the list of tuples. Then the hashlength is computed as

$$(1+2) + (1) + (1+2) + (1+2+3).$$

```

*****/

```

```

float objfunc (candidate, length, words, nw)
Chromosome   candidate;
int          length;
char         words[][MXLNGTH];
int          nw;
{

```

```

int    i, fscount=0, fs[MXLNGTH], j, overlap;
long   hashlength, find_average_hashlength();
float  fitness;
char   tuples[MAXNUMBEROFTUPLES][MXLNGTH];

/* Construct the set of tuples that are represented by the chromosome
   This is done by picking characters from the word list, at positions indicative
   of the presence of 1's in the chromosome */

for (i = 0; i < length; i++)
    if (candidate[i] == '1') {
        fs [fscount++] = i;
        if (fscount >= MXLNGTH) {
            printf ("\nThe size of the tuple has exceeded the maximum limit !.");
            printf ("\nAborting process.");
            exit (1);
        }
    }
for (i = 0; i < nw; i++)
    {
        for (j = 0; j < fscount; j++)
            tuples[i+1][j] = words[i][fs[j]];
        tuples[i+1][j] = '\0';
    }

heap_sort (tuples, nw);

hashlength = find_average_hashlength (tuples, nw);

/* RestrainLength is used in the method of selecting
   the better tuple among two tuples with the same fitness value but
   different tuple size.

   RestrainLength keeps track of the maximum fitness value for each
   tuple length size. If there exists a smaller tuple for which the
   fitness is = to the fitness of this chromosome, then this chromosome
   is undesirable and hence given a very low fitness. */

if (hashlength < RestrainLength[fscount])
    RestrainLength[fscount] = hashlength;

```

```

for (i = 1; i < fscount; i++) {
    if (RestrainingLength[fscount] == RestrainingLength[i]) {
        RestrainingLength[fscount] = 80000L;
        fitness = 1.0/80000L;
        return fitness;
    }
}

```

```

fitness = (float)nw/haslength;

```

```

return fitness;
}

```

```

/*****

```

Computes the hashlength of the tuples. T is list of tuples, and n is the number of tuples in this list.

The function first identifies groups from the list, such that each group consists of one or more identical tuples and no two groups have any tuples in common.

Then the hashlength is computed as sum of all sumseries for each group. sumseries for a group is the natural sum of its size i.e. if group size is 4, then sumseries for this group computes to 1+2+3+4.

```

*****/

```

```

long find_average_hashlength (T,n)
char T[][MXLNGTH];
int n;
{
    int i, overlap = 1;
    long sum, sumseries();
    char comparewith[MXLNGTH];

    sum = 0;
    strcpy (comparewith, T[1]);
    for (i = 2; i <= n; i++) {
        if (strcmp (comparewith, T[i]) == 0)
            overlap++;
        else {
            strcpy (comparewith, T[i]);
            sum += sumseries(overlap);
        }
    }
}

```

```

        overlap = 1;
    }
}
sum += sumseries(overlap);
return sum;
}

/*****
Computes the natural sum of its argument. If argument
is 5 then this function computes the natural sum
1+2+3+4+5.
*****/
long sumseries (uplimit)
int uplimit;
{
int i;
long sum;
sum = 0;
for (i = 1; i <= uplimit; i++)
    sum += i;
return sum;
}

/*****
Prints the list of tuples T. n is the number of tuples
in the list
*****/
printlist (T, n)
char T[][MXLNGTH];
int n;
{
int i;
printf ("\nThe tuples : \n");
for (i = 1; i <= n; i++) {
    printf ("%s ", T[i]);
    if (i%5 == 0) printf ("\n");
}
}

/*****
Splits the tuples into the groups by use of sorting.
*****/
heap_sort (T, n)
char T[][MXLNGTH];

```

```

int n;
{
int i;
char tmp[MXLNGTH];

make_heap (T, n);
for (i = n; i >= 2; i--) {
    strcpy (tmp,T[1]); strcpy (T[1],T[i]); strcpy (T[i],tmp);
    sift_down (T, 1, i-1);
}
}

make_heap (T, n)
char T[][MXLNGTH];
int n;
{
int i;

for (i = n/2; i >= 1; i--)
    sift_down (T, i, n);
}

sift_down (T, i, n)
char T[][MXLNGTH];
int i, n;
{
int k, j;
char tmp[MXLNGTH];

k = i;
do {
    j = k;
    if ((2*j <= n) && (strcmp(T[2*j], T[k]) > 0)) k = 2*j;
    if ((2*j < n) && (strcmp (T[2*j+1],T[k]) > 0)) k = 2*j + 1;
    strcpy (tmp,T[k]);
    strcpy (T[k],T[j]);
    strcpy (T[j],tmp);
} while (j != k);
}

```

APPENDIX C

The Configuration file used by the libGA implementation of the genetic algorithm called *tuple-app*.

```
#=====
# (c) Copyright Arthur L. Corcoran, 1992, 1993. All rights reserved.
#
# Genetic Algorithm configuration file
#=====
#-----
# User data file
# This information is not used by the GA, however, it is a convenient
# way to input a data file name or other information to your application.
#-----
# user_data datafile

#-----
# Seed for random number generator
#
# Usage: rand_seed my_pid
#       rand_seed number
#
# my_pid = use system pid as random seed
# number = seed for random number generator, a positive integer
#
# DEFAULT: rand_seed 1
#-----
# rand_seed my_pid
# rand_seed 1

#-----
# The data type of the allele
#
# Usage: datatype [bit | int | int_perm | real]
#
# bit    = bit string
# int    = integers
# int_perm = permutation of integers
# real   = real numbers
#
# DEFAULT: int_perm
```

```
#-----  
    datatype bit  
# datatype int  
# datatype int_perm  
# datatype real  
  
#-----  
# How to initialize the pool  
#  
# Usage: initpool [random | from_file filename | interactive]  
#  
#   random      = generate at random based on  
#                 datatype, chrom_len, & pool_size  
#   from_file   = read from a file  
#   filename    = the name of the file to read from  
#   interactive = read from stdin  
#  
# DEFAULT: initpool random  
#-----  
# initpool random  
# initpool from_file  initpool.dat  
# initpool interactive  
  
#-----  
# Chromosome length, needed when "initpool random" selected  
#  
# Usage: chrom_len length  
#  
#   length = chromosome length, a positive integer  
#  
# DEFAULT: chrom_len 10  
#-----  
# chrom_len 25  
  
#-----  
# Pool size, needed when "initpool random" selected  
#  
# Usage: pool_size size  
#  
#   size = pool size, a positive integer  
#  
# DEFAULT: 100  
#-----  
# pool_size 200
```



```

#-----
# When to stop the GA
#
# Convergence means when the variance = 0, or equivalently, when
# all the fitness values in the pool are identical.
#
# Iterations means the number of generations for the generational model
# and the number of trials for the steady state model. Numbers must
# be given as positive integers. It takes roughly pool_size/2
# iterations of the steady state model to equal one iteration of
# the generational model.
#
# Usage: stop_after convergence
#       stop_after number [use_convergence | ignore_convergence]
#
# convergence      - stop when the GA converges
# number           - stop after specified number of iterations
# use_convergence  - will stop early if GA converges (default)
# ignore_convergence - WILL NOT stop early even if GA converges
#
# DEFAULT: stop_after convergence
#-----
# stop_after convergence
# stop_after 5
# stop_after 500 use_convergence
# stop_after 500 ignore_convergence

#-----
# GA Type:
#
# Usage: ga [generational | steady_state]
#
# generational = generational GA
# steady_state = steady-state GA
#
# WARNING: This directive has the following side effects:
#
#       GA type           Directives set as a side effect
#       -----           -
#       generational     selection      roulette
#                       replacement    append
#                       rp_interval    1
#

```

```

#     steady-state   selection   rank_biased
#                   replacement  by_rank
#                   rp_interval  100
#
# DEFAULT: ga generational
#-----
# ga generational      # most commonly used
# ga steady_state     # used by Genitor
#-----
# Generation gap:
#
# The generation gap represents a percentage of the population to copy
# (clone) to the new pool at each generation. This only makes sense in
# a GA with two pools as in the generational model. A gap of 0.0
# is the traditional generational algorithm. As the gap increases,
# it becomes more like a steady-state algorithm. A gap of 1.0
# essentially disables crossover since only reproduction occurs.
#
# Usage: gap number
#
# number = generation gap, valid range = [0.0 .. 1.0]
#
# DEFAULT: gap 0.0
#-----
# gap 0.3
#-----
# Selection method:
#
# Usage: selection [roulette | rank_biased | uniform_random]
#
# roulette      = Roulette wheel
# rank_biased   = Ranked, biased selection as in Genitor
# uniform_random = Pick one at random
#
# DEFAULT: selection roulette
#-----
# selection roulette      # use with generational GA
# selection rank_biased   # use with steady-state GA
# selection uniform_random # experimental
#-----
# Selection bias

```

```

#
# Usage: bias number
#
#   number = selection bias, valid range = [1.0 .. 2.0]
#           Only used for rank_biased selection
#
# DEFAULT: bias 1.8
#-----
# bias 1.1

#-----
# Crossover method:
#
# Usage: crossover [simple | uniform | order1 | order2 | position | cycle |
#                 pmx | uox | rox | asexual]
#
# simple   = children get alternate "halves" of parents
# uniform  = alleles swapped uniformly
# order1   = order based
# order2   = order based
# position = order based
# cycle    = order based
# pmx      = order based
# uox      = uniform order
# rox      = relative order
# asexual  = swap two alleles
#
# DEFAULT: crossover order1
#-----
crossover simple
# crossover uniform
# crossover order1      # use ony with integer permutations
# crossover order2     # use ony with integer permutations
# crossover position    # use ony with integer permutations
# crossover cycle       # use ony with integer permutations
# crossover pmx         # use ony with integer permutations
# crossover uox         # use ony with integer permutations
# crossover rox         # use ony with integer permutations
# crossover asexual

#-----
# Crossover Rate
#
# Usage: x_rate number

```

```

#
# number = crossover rate (percentage), valid range = [0.0 .. 1.0]
#       A crossover rate of 0.0 disables crossover
#
# DEFAULT: x_rate 1.0
#-----
# x_rate 0.6

#-----
# Mutation method:
#
# Usage: mutation [simple_invert | simple_random | swap]
#
# simple_invert = invert a bit
# simple_random = random bit value
# swap          = swap two alleles
#
# DEFAULT: mutation swap
#-----
# mutation simple_invert    # use only with bits
# mutation simple_random    # use only with bits
# mutation swap             # use with any datatype

#-----
# Mutation Rate
#
# Usage: mu_rate number
#
# number = mutation rate (percentage), valid range = [0.0 .. 1.0]
#       A mutation rate of 0.0 disables mutation
#
# DEFAULT: mu_rate 0.0
#-----
# mu_rate 0.1

#-----
# Replacement method:
#
# Usage: replacement [append | by_rank | first_weaker | weakest]
#
# append      = append to new pool, as in generational GA
# by_rank     = insert in sorted order, as in Genitor
# first_weaker = replace first weaker found in linear scan of pool
# weakest     = replace weakest member of the pool

```

```

#
# DEFAULT: replacement append
#-----
# replacement append      # use with roulette (generational GA)
# replacement by_rank     # use with rank_biased (steady-state GA)
# replacement first_weaker # experimental
# replacement weakest     # experimental

#-----
# Objective of GA:
#
# Usage: objective [minimize | maximize]
#
# minimize = minimize evaluation function
# maximize = maximize evaluation function
#
# DEFAULT: objective minimize
#-----
# objective minimize
# objective maximize

#-----
# Elitism
#
# Elitism has two actions. For a generational GA, elitism makes two copies
# of the best performer in the old pool and places them in the new
# pool, thus ensuring the most fit chromosome survives. The other action
# works with both models. In this case, elitism picks the best two
# chromosomes from the parents and children. Thus, if a child is not as
# fit as either parent, it will not be placed in the new pool. Selecting
# elitism in LibGA performs both actions.
#
# Usage: elitism [true | false]
#
# true  = ensure best members survive until next generation
# false = no guarantee best will survive
#
# DEFAULT: elitism true
#-----
# elitism true
# elitism false

#-----
# Report type

```

```

#
# Usage: rp_type [none | minimal | short | long]
#
# none    = output nothing
# minimal = output configuration and final result
# short   = output minimal + statistics only
# long    = output short + dump pool
#
# DEFAULT: rp_type short
#-----
# rp_type none
# rp_type minimal
# rp_type short
# rp_type long

#-----
# Report interval
#
# Usage: rp_interval number
#
# number = interval between reports, a positive integer
#
# DEFAULT: rp_interval 1
#-----
# rp_interval 10

#-----
# Output report filename
#
# Usage: rp_file file_name [file_mode]
# =====*/
#
# file_name = name of report file
# file_mode = optional file mode for fopen()
#   a      = append (DEFAULT)
#   w      = overwrite
#
# DEFAULT: (write to stdout)
#-----
# rp_file ga.out
# rp_file ga.out a
# rp_file ga.out w

```

The source code of the libGA implementation of the genetic algorithm called *tuple-app*

```

#include "ga.h"          /* code for the functions relating to the genetic
                        operations. (Part of the libGA application) */
#include "const.h"      /* Contains constants used by the application */
#include "eval.h"       /* Fitness evaluation related functions */

int length, nw;        /* length stores the length of the words
                        and nw stores the number of words contained
                        in the input data file */

char words[MAXNUMBEROFTUPLES][MXLNGTH];
                        /* The list of words from the input data file */

int obj_fun();        /*--- Forward declaration of the fitness function ---*/

/*-----
| main() : The main program runs the genetic algorithm. The name of the
|           input data file is "datafile". Two runs are performed: one using
|           the elitism and the other with elitism turned off.
|           The configuration settings for this application are in the file
|           called "tuple-app.cfg"
-----*/

main(argc, argv)
    int argc;
    char *argv[];
{
    GA_Info_Ptr ga_info;

    /*--- Initialize the genetic algorithm ---*/
    ga_info = GA_config("tuple-app.cfg", obj_fun);

    /*--- Read the input word list ---*/
    read_words ("datafile");

    /*--- Set chromosome length to the word length of the input word list ---*/
    ga_info->chrom_len = length;

    /*--- Use elitism ---*/
    ga_info->elitist=1;
    /*--- Run the Genetic algorithm ---*/

```

```

GA_run(ga_info);

/*--- Reset the genetic algorithm for the second run */
GA_reset (ga_info, "tuple-app.cfg");
/*--- Turn elitism off ---*/
ga_info->elitist=0;
/*--- Run the Genetic algorithm ---*/
GA_run(ga_info);

}

/*-----
| obj_fun() - user specified objective function
-----*/

```

```

/*****

```

The FITNESS Evaluation Module

The objfunc () takes a chromosome as an argument and assigns it a fitness value in the range 0..1. The higher the fitness value the closer the chromosome is to the solution. For fitness value of 1, the chromosome represents the solution.

The chromosome represents a set of tuples. The fitness value gives a measure of the number of duplicate entries existing in the list of tuples. This is computed as an average hashlength = $nw/hashlength$.

for example : if the chromosome 0101 represents the tuples : aa ab cd aa ba cd ba ba, then a frequency count of this list will tell us that there are 2 aa's, 1 ab, 2 cd's, and 3 ba's in the list of tuples. Then the hashlength is computed as

$$(1+2) + (1) + (1+2) + (1+2+3).$$

and the fitness is evaluated as

$$(\text{number of words}) / \text{hashlength}$$

```

*****/

```

```

int obj_fun (chrom)
Chrom_Ptr chrom;
{
int      i, fscount=0, fs[MXLNGTH], j, overlap;
long     hashlength, find_average_hashlength();
float    fitness;
char     tuples[MAXNUMBEROFTUPLES][MXLNGTH];

```



```

/* Construct the set of tuples that are represented by the chromosome
   This is done by picking characters from the word list, at positions
   indicative of the presence of 1's in the chromosome */

for (i = 0; i < chrom->length; i++)
    if (chrom->gene[i] == 1) {
        fs [fscount++] = i;
        if (fscount >= MXLNGTH) {
            printf ("\nThe size of the tuple has exceeded the maximum limit !.");
            printf ("\nAborting process.");
            exit (1);
        }
    }
for (i = 0; i < nw; i++)
    {
        for (j = 0; j < fscount; j++)
            tuples[i+1][j] = words[i][fs[j]];
        tuples[i+1][j] = '\0';
    }

/* This function computes the frequency count of the tuples. First the tuples
   are split into groups of identical tuples. And then the number of tuples
   in each group gives is used to find the hashlength */
make_groups (tuples, nw);

/* compute the hashlength */
hashlength = find_average_hashlength (tuples, nw);

/* compute the fitness */
chrom->fitness = (float)nw/hashlength;
}

```

```

/*****

```

The read_words () function

The input data, the word list, is read from the input file.

```

*****/

```

```

read_words (filename)
char    *filename;
{
int     i;
char    ch;
FILE    *fp;

```

```

/* Opening the input file that contains the list of words */
if ((fp = fopen(filename, "r")) == NULL) {
    printf ("\nError in opening the input file containing the words.\n");
    exit (1);
}

```

```

/* Read in the header section of the input file, which contains the
   the parameters : nw (number of words ), lchrom (word length),
   and the letters (the alphabet comprising the words) */
fscanf (fp,"%d %d ",&nw ,&length);
i = 0;
while ((ch=getc(fp)) != '\n') ;
/* Read in the words from the input file*/
i = 0;
while (fscanf (fp, "%s", words[i]) != EOF) i++;
fclose (fp);
}

```

```

/*****
Computes the hashlength of the tuples. T is list of
tuples, and n is the number of tuples in this list.

```

The function first identifies groups from the list, such that each group consists of one or more identical tuples and no two groups have any tuples in common.

Then the hashlength is computed as sum of all sumseries for each group. sumseries for a group is the natural sum of its size i.e. if group size is 4, say, then sumseries for this group computes to 1+2+3+4.

```

*****/
long find_average_hashlength (T,n)
char T[][MXLNGTH];
int n;
{
    int i, overlap = 1;
    long sum, sumseries();
    char comparewith[MXLNGTH];

    sum = 0;
    strcpy (comparewith, T[1]);
    for (i = 2; i <= n; i++) {

```

```

    if (strcmp (comparewith, T[i]) == 0)
        overlap++;
    else {

        strcpy (comparewith, T[i]);
        sum += sumseries(overlap);
        overlap = 1;
    }
}
sum += sumseries(overlap);
return sum;
}

/*****
Computes the natural sum of its argument. If argument
is 5 then this function computes the natural sum
1+2+3+4+5.
*****/
long sumseries (uplimit)
int uplimit;
{
int i;
long sum;
sum = 0;
for (i = 1; i <= uplimit; i++)
    sum += i;
return sum;
}

```

/* The code for the remaining functions used are same as that used in the C language implementation : *genrph* */

VITA

Ravi B. Mandadi

Candidate for the degree of

Master of Science

Thesis: A GENETIC ALGORITHM TO EXTRACT N-TUPLES FROM A GIVEN SET OF WORDS

Major Field: Computer Science

Biographical:

Personal Data: Born in Guntur District, Hyderabad, India, on July 24, 1967, son of Venkateswarlu and Rajyalakshmi Mandadi.

Education: Received Bachelor of Science degree in Electrical Engineering from Osmania University, Hyderabad, India in July 1989. Completed the requirements for the Master of Science degree with a major in Computer Science at Oklahoma State University in May 1995.

Experience: Employed by Oklahoma State University, Department of Agricultural Engineering as a graduate research assistant; Oklahoma State University, Department of Agricultural Engineering, 1991 to 1994.